Configuring Infinispan 10.0

Table of Contents

1. Infinispan Caches . . . 1
   1.1. Cache Interface . . . 1
   1.2. Cache Managers . . . 1
   1.3. Cache Containers . . . 1
   1.4. Cache Modes . . . 2
      1.4.1. Cache Mode Comparison . . . 2
2. Local Caches . . . 4
   2.1. Simple Caches . . . 4
3. Clustered Caches . . . 6
   3.1. Invalidation Mode . . . 6
   3.2. Replicated Caches . . . 8
   3.3. Distributed Caches . . . 10
      3.3.1. Read consistency . . . 12
      3.3.2. Key Ownership . . . 13
      3.3.3. Zero Capacity Node . . . 14
      3.3.4. Hashing Configuration . . . 14
      3.3.5. Initial cluster size . . . 15
      3.3.6. L1 Caching . . . 15
      3.3.7. Server Hinting . . . 17
      3.3.8. Key affinity service . . . 17
   3.4. Scattered Caches . . . 22
   3.5. Asynchronous Communication with Clustered Caches . . . 23
      3.5.1. Asynchronous Communications . . . 23
      3.5.2. Asynchronous API . . . 23
      3.5.3. Return Values in Asynchronous Communication . . . 23
4. Configuring Caches Declaratively . . . 25
   4.1. Infinispan subsystem . . . 25
      4.1.1. Containers . . . 25
      4.1.2. Cache declarations . . . 25
   4.2. Locking . . . 26
   4.3. Loaders and Stores . . . 26
   4.4. State Transfer . . . 27
   4.5. Declarative Cache Configuration . . . 28
   4.6. Cache configuration templates . . . 29
   4.7. Cache configuration wildcards . . . 31
   4.8. XInclude support . . . 31
   4.9. Declarative configuration reference . . . 31
5. Configuring Caches Programmatically . . . 33

5.1. CacheManager and ConfigurationBuilder API . . . 33
5.2. ConfigurationBuilder Programmatic Configuration API . . . 34
5.2.1. Enabling JMX MBeans and statistics . . . 34
5.2.2. Configuring thread pools . . . 35
5.2.3. Configuring transactions and locking . . . 35
5.2.4. Configuring cache stores . . . 36
5.2.5. Advanced programmatic configuration . . . 36
6. Setting Up Cluster Transport . . . 38
6.1. Getting Started with Default Stacks . . . 38
6.1.1. Default JGroups Stacks . . . 39
6.1.2. Default JGroups Stacks . . . 39
6.2. Using Inline JGroups Stacks . . . 41
6.3. Adjusting and Tuning JGroups Stacks . . . 42
6.3.1. Stack Combine Attribute . . . 43
6.4. Using JGroups Stacks in External Files . . . 44
6.5. Tuning JGroups Stacks with System Properties . . . 44
6.5.1. System Properties for Default JGroups Stacks . . . 45
6.6. Using Custom JChannels . . . 46
7. Configuring Cluster Discovery . . . 47
7.1. TCPPING . . . 47
7.2. Gossip Router . . . 47
7.3. DNS_PING . . . 48
7.4. KUBE_PING . . . 48
7.5. NATIVE_S3_PING . . . 49
7.6. JDBC_PING . . . 50
7.7. AZURE_PING . . . 50
7.8. GOOGLE2_PING . . . 50

Chapter 1. Infinispan Caches

Infinispan caches provide flexible, in-memory data stores that you can configure to suit use cases such as:

- boosting application performance with high-speed local caches.
- optimizing databases by decreasing the volume of write operations.
- providing resiliency and durability for consistent data across clusters.

1.1. Cache Interface

`Cache<K,V>` is the central interface for Infinispan and extends `java.util.concurrent.ConcurrentMap`.

Cache entries are highly concurrent data structures in `key:value` format that support a wide and configurable range of data types, from simple strings to much more complex objects.

1.2. Cache Managers

Infinispan provides a `CacheManager` interface that lets you create, modify, and manage local or clustered caches. Cache Managers are the starting point for using Infinispan caches.

There are two `CacheManager` implementations:

`EmbeddedCacheManager`
: Entry point for caches when running Infinispan inside the same Java Virtual Machine (JVM) as the client application, which is also known as Library Mode.

`RemoteCacheManager`
: Entry point for caches when running Infinispan as a remote server in its own JVM. When it starts running, `RemoteCacheManager` establishes a persistent TCP connection to a Hot Rod endpoint on a Infinispan server.

Both embedded and remote `CacheManager` implementations share some methods and properties. However, semantic differences do exist between `EmbeddedCacheManager` and `RemoteCacheManager`.

1.3. Cache Containers

Cache containers declare one or more local or clustered caches that a Cache Manager controls.

Cache container declaration

```
<cache-container name="clustered" default-cache="default">
  ...
</cache-container>
```

1.4. Cache Modes

Infinispan Cache Managers can create and control multiple caches that use different modes. For example, you can use the same Cache Manager for local caches, distributes caches, and caches with invalidation mode.

Local Caches

Infinispan runs as a single node and never replicates read or write operations on cache entries.

Clustered Caches

Infinispan instances running on the same network can automatically discover each other and form clusters to handle cache operations.

Invalidation Mode

Rather than replicating cache entries across the cluser, Infinispan evicts stale data from all nodes whenever operations modify entries in the cache. Infinispan performs local read operations only.

Replicated Caches

Infinispan replicates each cache entry on all nodes and performs local read operations only.

Distributed Caches

Infinispan stores cache entries across a subset of nodes and assigns entries to fixed owner nodes. Infinispan requests read operations from owner nodes to ensure it returns the correct value.

Scattered Caches

Infinispan stores cache entries across a subset of nodes. By default Infinispan assigns a primary owner and a backup owner to each cache entry in scattered caches. Infinispan assigns primary owners in the same way as with distributed caches, while backup owners are always the nodes that initiate the write operations. Infinispan requests read operations from at least one owner node to ensure it returns the correct value.

1.4.1. Cache Mode Comparison

The cache mode that you should choose depends on the qualities and guarantees you need for your data.

The following table summarizes the primary differences between cache modes:

	Simple	Local	Invalidation	Replicated	Distributed	Scattered
Clustered	**No**	**No**	**Yes**	**Yes**	**Yes**	**Yes**
Read performance	**Highest** (local)	**High** (local)	**High** (local)	**High** (local)	**Medium** (owners)	**Medium** (primary)

	Simple	Local	Invalidation	Replicated	Distributed	Scattered
Write performance	**Highest** (local)	**High** (local)	**Low** (all nodes, no data)	**Lowest** (all nodes)	**Medium** (owner nodes)	**Higher** (single RPC)
Capacity	**Single node**	**Single node**	**Single node**	**Smallest node**	**Cluster** (sum_(i=1)^"nodes""node_capacity")/"owners"	**Cluster** (sum_(i=1)^"nodes""node_capacity")/"2"
Availability	**Single node**	**Single node**	**Single node**	**All nodes**	**Owner nodes**	**Owner nodes**
Features	**No TX, persistence, indexing**	**All**	**All**	**All**	**All**	**No TX**

Chapter 2. Local Caches

While Infinispan is particularly interesting in clustered mode, it also offers a very capable local mode. In this mode, it acts as a simple, in-memory data cache similar to a `ConcurrentHashMap`.

But why would one use a local cache rather than a map? Caches offer a lot of features over and above a simple map, including write-through and write-behind to a persistent store, eviction of entries to prevent running out of memory, and expiration.

Infinispan's `Cache` interface extends JDK's `ConcurrentMap` — making migration from a map to Infinispan trivial.

Infinispan caches also support transactions, either integrating with an existing transaction manager or running a separate one. Local caches transactions have two choices:

1. When to lock? **Pessimistic locking** locks keys on a write operation or when the user calls `AdvancedCache.lock(keys)` explicitly. **Optimistic locking** only locks keys during the transaction commit, and instead it throws a `WriteSkewCheckException` at commit time, if another transaction modified the same keys after the current transaction read them.
2. Isolation level. We support **read-committed** and **repeatable read**.

2.1. Simple Caches

Traditional local caches use the same architecture as clustered caches, i.e. they use the interceptor stack. That way a lot of the implementation can be reused. However, if the advanced features are not needed and performance is more important, the interceptor stack can be stripped away and simple cache can be used.

So, which features are stripped away? From the configuration perspective, simple cache does not support:

- transactions and invocation batching
- persistence (cache stores and loaders)
- custom interceptors (there's no interceptor stack!)
- indexing
- transcoding
- store as binary (which is hardly useful for local caches)

From the API perspective these features throw an exception:

- adding custom interceptors
- Distributed Executors Framework

So, what's left?

- basic map-like API

- cache listeners (local ones)
- expiration
- eviction
- security
- JMX access
- statistics (though for max performance it is recommended to switch this off using statistics-available=false)

Declarative configuration

```
<local-cache name="mySimpleCache" simple-cache="true">
    <!-- expiration, eviction, security... -->
</local-cache>
```

Programmatic configuration

```
CacheManager cm = getCacheManager();
ConfigurationBuilder builder = new ConfigurationBuilder().simpleCache(true);
cm.defineConfiguration("mySimpleCache", builder.build());
Cache cache = cm.getCache("mySimpleCache");
```

Simple cache checks against features it does not support, if you configure it to use e.g. transactions, configuration validation will throw an exception.

Chapter 3. Clustered Caches

Clustered caches store data across multiple Infinispan nodes using JGroups technology as the transport layer to pass data across the network.

3.1. Invalidation Mode

You can use Infinispan in invalidation mode to optimize systems that perform high volumes of read operations. A good example is to use invalidation to prevent lots of database writes when state changes occur.

This cache mode only makes sense if you have another, permanent store for your data such as a database and are only using Infinispan as an optimization in a read-heavy system, to prevent hitting the database for every read. If a cache is configured for invalidation, every time data is changed in a cache, other caches in the cluster receive a message informing them that their data is now stale and should be removed from memory and from any local store.

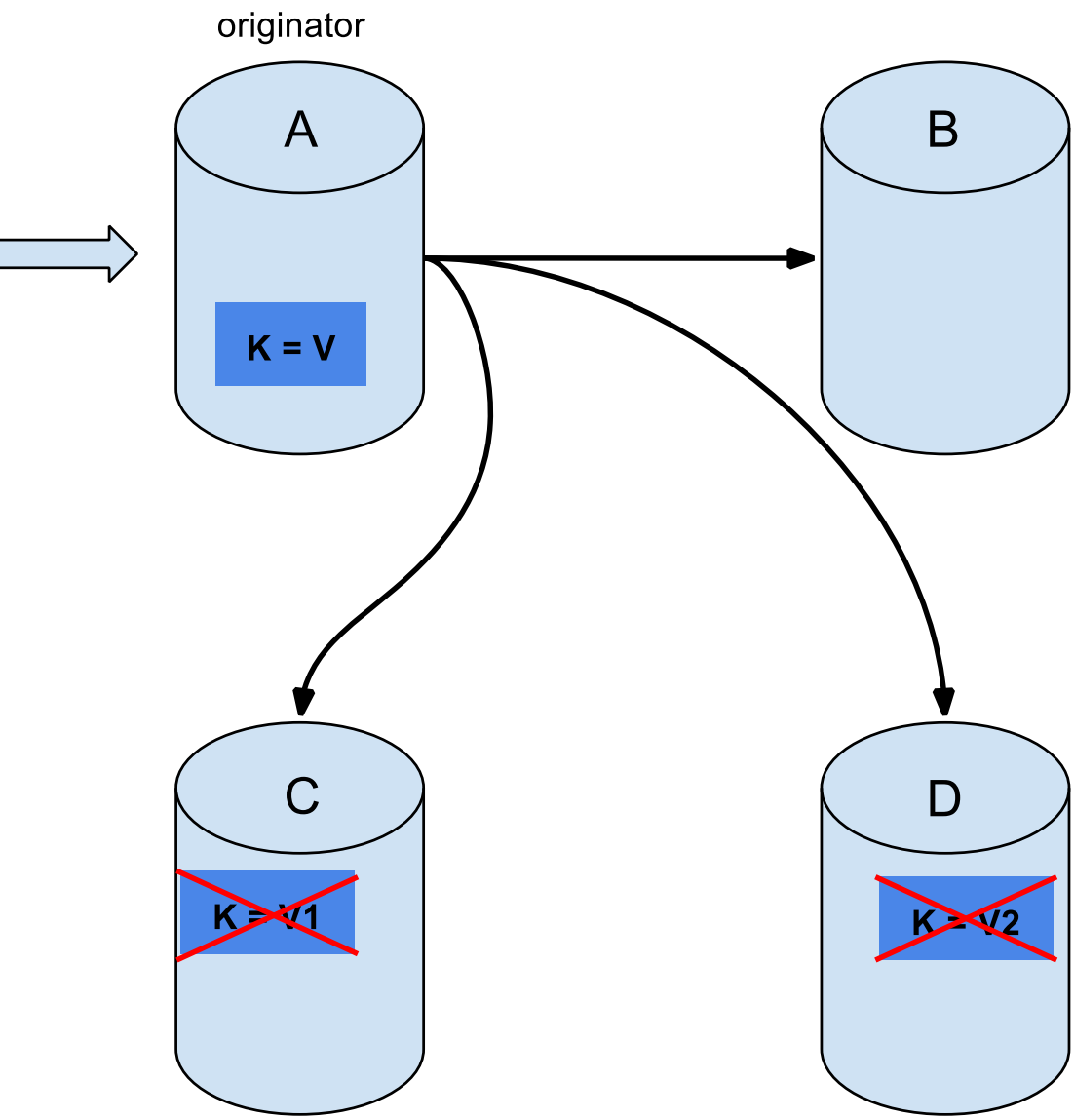

Figure 1. Invalidation mode

Sometimes the application reads a value from the external store and wants to write it to the local cache, without removing it from the other nodes. To do this, it must call `Cache.putForExternalRead(key, value)` instead of `Cache.put(key, value)`.

Invalidation mode can be used with a shared cache store. A write operation will both update the shared store, and it would remove the stale values from the other nodes' memory. The benefit of this is twofold: network traffic is minimized as invalidation messages are very small compared to replicating the entire value, and also other caches in the cluster look up modified data in a lazy manner, only when needed.

Never use invalidation mode with a **local** store. The invalidation message will not remove entries in the local store, and some nodes will keep seeing the stale value.

An invalidation cache can also be configured with a special cache loader, `ClusterLoader`. When `ClusterLoader` is enabled, read operations that do not find the key on the local node will request it from all the other nodes first, and store it in memory locally. In certain situation it will store stale values, so only use it if you have a high tolerance for stale values.

Invalidation mode can be synchronous or asynchronous. When synchronous, a write blocks until all nodes in the cluster have evicted the stale value. When asynchronous, the originator broadcasts invalidation messages but doesn't wait for responses. That means other nodes still see the stale value for a while after the write completed on the originator.

Transactions can be used to batch the invalidation messages. Transactions acquire the key lock on the primary owner. To find more about how primary owners are assigned, please read the Key Ownership section.

- With pessimistic locking, each write triggers a lock message, which is broadcast to all the nodes. During transaction commit, the originator broadcasts a one-phase prepare message (optionally fire-and-forget) which invalidates all affected keys and releases the locks.
- With optimistic locking, the originator broadcasts a prepare message, a commit message, and an unlock message (optional). Either the one-phase prepare or the unlock message is fire-and-forget, and the last message always releases the locks.

3.2. Replicated Caches

Entries written to a replicated cache on any node will be replicated to all other nodes in the cluster, and can be retrieved locally from any node. Replicated mode provides a quick and easy way to share state across a cluster, however replication practically only performs well in small clusters (under 10 nodes), due to the number of messages needed for a write scaling linearly with the cluster size. Infinispan can be configured to use UDP multicast, which mitigates this problem to some degree.

Each key has a primary owner, which serializes data container updates in order to provide consistency. To find more about how primary owners are assigned, please read the Key Ownership section.

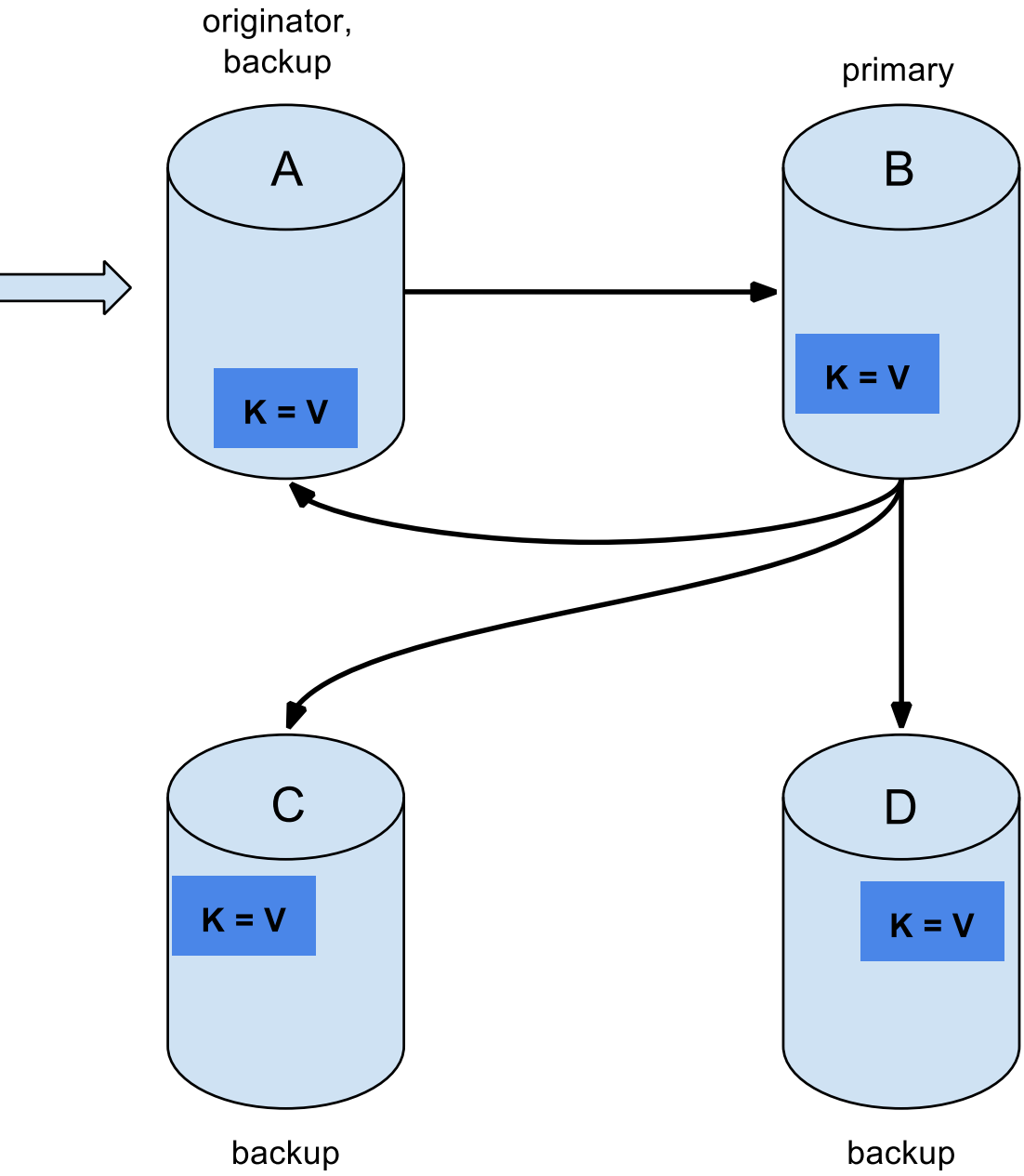

Figure 2. Replicated mode

Replicated mode can be synchronous or asynchronous.

- Synchronous replication blocks the caller (e.g. on a `cache.put(key, value)`) until the modifications have been replicated successfully to all the nodes in the cluster.
- Asynchronous replication performs replication in the background, and write operations return immediately. Asynchronous replication is not recommended, because communication errors, or errors that happen on remote nodes are not reported to the caller.

If transactions are enabled, write operations are not replicated through the primary owner.

- With pessimistic locking, each write triggers a lock message, which is broadcast to all the nodes. During transaction commit, the originator broadcasts a one-phase prepare message and an unlock message (optional). Either the one-phase prepare or the unlock message is fire-and-forget.
- With optimistic locking, the originator broadcasts a prepare message, a commit message, and an unlock message (optional). Again, either the one-phase prepare or the unlock message is fire-and-forget.

3.3. Distributed Caches

Distribution tries to keep a fixed number of copies of any entry in the cache, configured as `numOwners`. This allows the cache to scale linearly, storing more data as nodes are added to the cluster.

As nodes join and leave the cluster, there will be times when a key has more or less than `numOwners` copies. In particular, if `numOwners` nodes leave in quick succession, some entries will be lost, so we say that a distributed cache tolerates `numOwners - 1` node failures.

The number of copies represents a trade-off between performance and durability of data. The more copies you maintain, the lower performance will be, but also the lower the risk of losing data due to server or network failures. Regardless of how many copies are maintained, distribution still scales linearly, and this is key to Infinispan's scalability.

The owners of a key are split into one **primary owner**, which coordinates writes to the key, and zero or more **backup owners**. To find more about how primary and backup owners are assigned, please read the Key Ownership section.

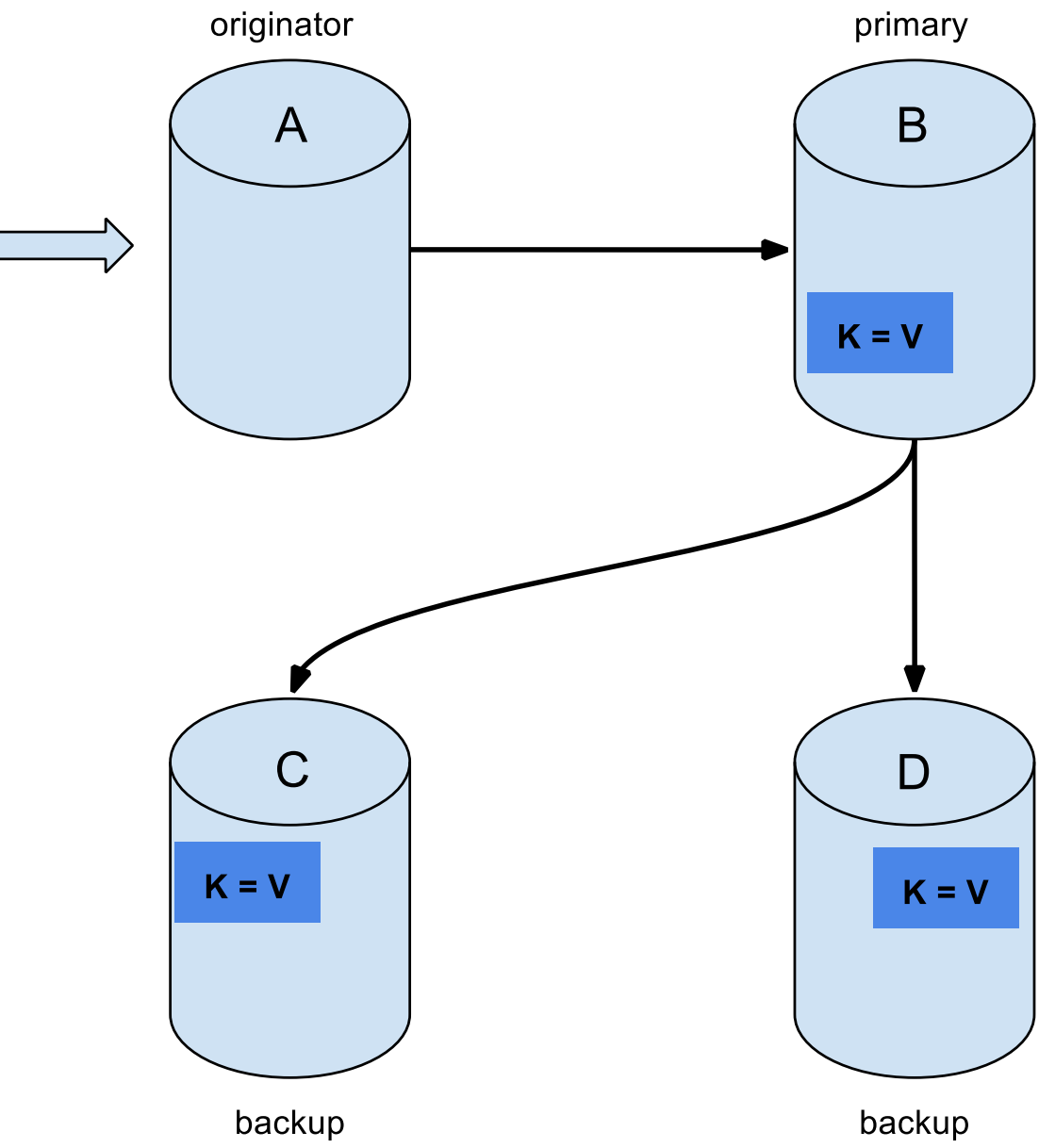

Figure 3. Distributed mode

A read operation will request the value from the primary owner, but if it doesn't respond in a reasonable amount of time, we request the value from the backup owners as well. (The `infinispan.stagger.delay` system property, in milliseconds, controls the delay between requests.) A read operation may require `0` messages if the key is present in the local cache, or up to `2 * numOwners` messages if all the owners are slow.

A write operation will also result in at most `2 * numOwners` messages: one message from the originator to the primary owner, `numOwners - 1` messages from the primary to the backups, and the corresponding ACK messages.

Cache topology changes may cause retries and additional messages, both for reads and for writes.

Just as replicated mode, distributed mode can also be synchronous or asynchronous. And as in replicated mode, asynchronous replication is not recommended because it can lose updates. In addition to losing updates, asynchronous distributed caches can also see a stale value when a thread writes to a key and then immediately reads the same key.

Transactional distributed caches use the same kinds of messages as transactional replicated caches, except lock/prepare/commit/unlock messages are sent only to the **affected nodes** (all the nodes that own at least one key affected by the transaction) instead of being broadcast to all the nodes in the cluster. As an optimization, if the transaction writes to a single key and the originator is the primary owner of the key, lock messages are not replicated.

3.3.1. Read consistency

Even with synchronous replication, distributed caches are not linearizable. (For transactional caches, we say they do not support serialization/snapshot isolation.) We can have one thread doing a single put:

```
cache.get(k) -> v1
cache.put(k, v2)
cache.get(k) -> v2
```

But another thread might see the values in a different order:

```
cache.get(k) -> v2
cache.get(k) -> v1
```

The reason is that read can return the value from **any** owner, depending on how fast the primary owner replies. The write is not atomic across all the owners — in fact, the primary commits the update only after it receives a confirmation from the backup. While the primary is waiting for the confirmation message from the backup, reads from the backup will see the new value, but reads from the primary will see the old one.

3.3.2. Key Ownership

Distributed caches split entries into a fixed number of segments and assign each segment to a list of owner nodes. Replicated caches do the same, with the exception that every node is an owner.

The first node in the list of owners is the **primary owner**. The other nodes in the list are **backup owners**. When the cache topology changes, because a node joins or leaves the cluster, the segment ownership table is broadcast to every node. This allows nodes to locate keys without making multicast requests or maintaining metadata for each key.

The `numSegments` property configures the number of segments available. However, the number of segments cannot change unless the cluster is restarted.

Likewise the key-to-segment mapping cannot change. Keys must always map to the same segments regardless of cluster topology changes. It is important that the key-to-segment mapping evenly distributes the number of segments allocated to each node while minimizing the number of segments that must move when the cluster topology changes.

You can customize the key-to-segment mapping by configuring a KeyPartitioner or by using the Grouping API.

However, Infinispan provides the following implementations:

SyncConsistentHashFactory

Uses an algorithm based on consistent hashing. Selected by default when server hinting is disabled.

This implementation always assigns keys to the same nodes in every cache as long as the cluster is symmetric. In other words, all caches run on all nodes. This implementation does have some negative points in that the load distribution is slightly uneven. It also moves more segments than strictly necessary on a join or leave.

TopologyAwareSyncConsistentHashFactory

Similar to `SyncConsistentHashFactory`, but adapted for Server Hinting. Selected by default when server hinting is enabled.

DefaultConsistentHashFactory

Achieves a more even distribution than `SyncConsistentHashFactory`, but with one disadvantage. The order in which nodes join the cluster determines which nodes own which segments. As a result, keys might be assigned to different nodes in different caches.

Was the default from version 5.2 to version 8.1 with server hinting disabled.

TopologyAwareConsistentHashFactory

Similar to *DefaultConsistentHashFactory*, but adapted for Server Hinting.

Was the default from version 5.2 to version 8.1 with server hinting enabled.

ReplicatedConsistentHashFactory

Used internally to implement replicated caches. You should never explicitly select this algorithm

in a distributed cache.

Capacity Factors

Capacity factors are another way to customize the mapping of segments to nodes. The nodes in a cluster are not always identical. If a node has 2x the memory of a "regular" node, configuring it with a `capacityFactor` of `2` tells Infinispan to allocate 2x segments to that node. The capacity factor can be any non-negative number, and the hashing algorithm will try to assign to each node a load weighted by its capacity factor (both as a primary owner and as a backup owner).

One interesting use case is nodes with a capacity factor of `0`. This could be useful when some nodes are too short-lived to be useful as data owners, but they can't use HotRod (or other remote protocols) because they need transactions. With cross-site replication as well, the "site master" should only deal with forwarding commands between sites and shouldn't handle user requests, so it makes sense to configure it with a capacity factor of `0`.

3.3.3. Zero Capacity Node

You might need to configure a whole node where the capacity factor is `0` for every cache, user defined caches and internal caches. When defining a zero capacity node, the node won't hold any data. This is how you declare a zero capacity node:

```
<cache-container zero-capacity-node="true" />
```

```
new GlobalConfigurationBuilder().zeroCapacityNode(true);
```

However, note that this will be true for distributed caches only. If you are using replicated caches, the node will still keep a copy of the value. Use only distributed caches to make the best use of this feature.

3.3.4. Hashing Configuration

This is how you configure hashing declaratively, via XML:

```
<distributed-cache name="distributedCache" owners="2" segments="100" capacity-factor=
"2" />
```

And this is how you can configure it programmatically, in Java:

```
Configuration c = new ConfigurationBuilder()
   .clustering()
      .cacheMode(CacheMode.DIST_SYNC)
      .hash()
         .numOwners(2)
         .numSegments(100)
         .capacityFactor(2)
   .build();
```

3.3.5. Initial cluster size

Infinispan's very dynamic nature in handling topology changes (i.e. nodes being added / removed at runtime) means that, normally, a node doesn't wait for the presence of other nodes before starting. While this is very flexible, it might not be suitable for applications which require a specific number of nodes to join the cluster before caches are started. For this reason, you can specify how many nodes should have joined the cluster before proceeding with cache initialization. To do this, use the `initialClusterSize` and `initialClusterTimeout` transport properties. The declarative XML configuration:

```
<transport initial-cluster-size="4" initial-cluster-timeout="30000" />
```

The programmatic Java configuration:

```
GlobalConfiguration global = new GlobalConfigurationBuilder()
   .transport()
       .initialClusterSize(4)
       .initialClusterTimeout(30000)
   .build();
```

The above configuration will wait for *4* nodes to join the cluster before initialization. If the initial nodes do not appear within the specified timeout, the cache manager will fail to start.

3.3.6. L1 Caching

When L1 is enabled, a node will keep the result of remote reads locally for a short period of time (configurable, 10 minutes by default), and repeated lookups will return the local L1 value instead of asking the owners again.

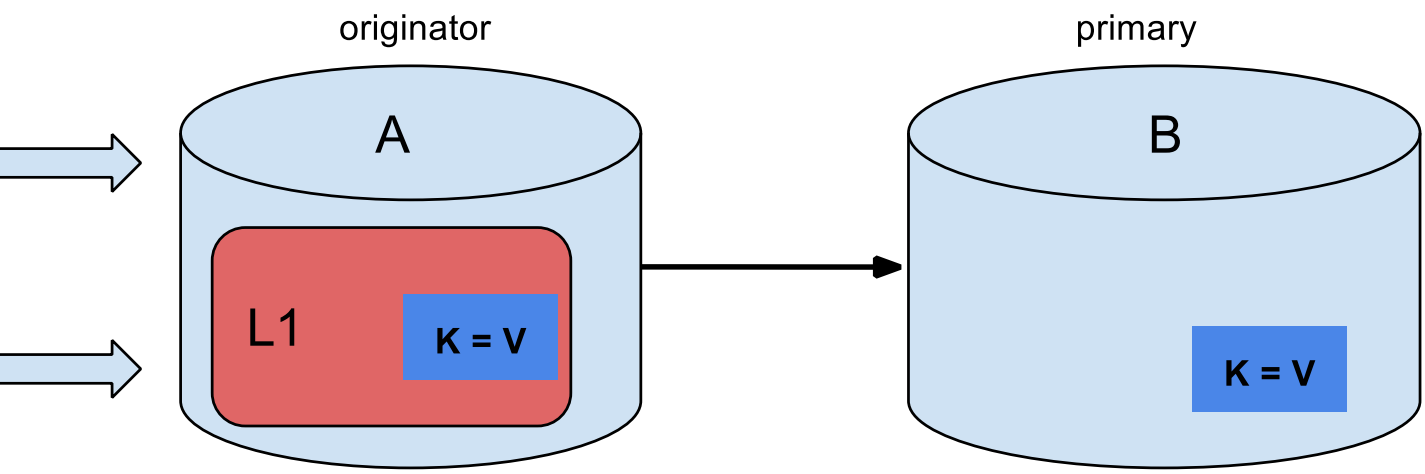

Figure 4. L1 caching

L1 caching is not free though. Enabling it comes at a cost, and this cost is that every entry update must broadcast an invalidation message to all the nodes. L1 entries can be evicted just like any other entry when the the cache is configured with a maximum size. Enabling L1 will improve performance for repeated reads of non-local keys, but it will slow down writes and it will increase memory consumption to some degree.

Is L1 caching right for you? The correct approach is to benchmark your application with and without L1 enabled and see what works best for your access pattern.

3.3.7. Server Hinting

The following topology hints can be specified:

Machine

This is probably the most useful, when multiple JVM instances run on the same node, or even when multiple virtual machines run on the same physical machine.

Rack

In larger clusters, nodes located on the same rack are more likely to experience a hardware or network failure at the same time.

Site

Some clusters may have nodes in multiple physical locations for extra resilience. Note that Cross site replication is another alternative for clusters that need to span two or more data centres.

All of the above are optional. When provided, the distribution algorithm will try to spread the ownership of each segment across as many sites, racks, and machines (in this order) as possible.

Configuration

The hints are configured at transport level:

```
<transport
    cluster="MyCluster"
    machine="LinuxServer01"
    rack="Rack01"
    site="US-WestCoast" />
```

3.3.8. Key affinity service

In a distributed cache, a key is allocated to a list of nodes with an opaque algorithm. There is no easy way to reverse the computation and generate a key that maps to a particular node. However, we can generate a sequence of (pseudo-)random keys, see what their primary owner is, and hand them out to the application when it needs a key mapping to a particular node.

API

Following code snippet depicts how a reference to this service can be obtained and used.

```
// 1. Obtain a reference to a cache
Cache cache = ...
Address address = cache.getCacheManager().getAddress();

// 2. Create the affinity service
KeyAffinityService keyAffinityService = KeyAffinityServiceFactory
.newLocalKeyAffinityService(
      cache,
      new RndKeyGenerator(),
      Executors.newSingleThreadExecutor(),
      100);

// 3. Obtain a key for which the local node is the primary owner
Object localKey = keyAffinityService.getKeyForAddress(address);

// 4. Insert the key in the cache
cache.put(localKey, "yourValue");
```

The service is started at step 2: after this point it uses the supplied *Executor* to generate and queue keys. At step 3, we obtain a key from the service, and at step 4 we use it.

Lifecycle

`KeyAffinityService` extends `Lifecycle`, which allows stopping and (re)starting it:

```
public interface Lifecycle {
   void start();
   void stop();
}
```

The service is instantiated through `KeyAffinityServiceFactory`. All the factory methods have an `Executor` parameter, that is used for asynchronous key generation (so that it won't happen in the caller's thread). It is the user's responsibility to handle the shutdown of this `Executor`.

The `KeyAffinityService`, once started, needs to be explicitly stopped. This stops the background key generation and releases other held resources.

The only situation in which `KeyAffinityService` stops by itself is when the cache manager with which it was registered is shutdown.

Topology changes

When the cache topology changes (i.e. nodes join or leave the cluster), the ownership of the keys generated by the `KeyAffinityService` might change. The key affinity service keep tracks of these topology changes and doesn't return keys that would currently map to a different node, but it won't do anything about keys generated earlier.

As such, applications should treat `KeyAffinityService` purely as an optimization, and they should

not rely on the location of a generated key for correctness.

In particular, applications should not rely on keys generated by `KeyAffinityService` for the same address to always be located together. Collocation of keys is only provided by the Grouping API.

The Grouping API

Complementary to Key affinity service, the grouping API allows you to co-locate a group of entries on the same nodes, but without being able to select the actual nodes.

How does it work?

By default, the segment of a key is computed using the key's `hashCode()`. If you use the grouping API, Infinispan will compute the segment of the group and use that as the segment of the key. See the Key Ownership section for more details on how segments are then mapped to nodes.

When the group API is in use, it is important that every node can still compute the owners of every key without contacting other nodes. For this reason, the group cannot be specified manually. The group can either be intrinsic to the entry (generated by the key class) or extrinsic (generated by an external function).

How do I use the grouping API?

First, you must enable groups. If you are configuring Infinispan programmatically, then call:

```
Configuration c = new ConfigurationBuilder()
   .clustering().hash().groups().enabled()
   .build();
```

Or, if you are using XML:

```
<distributed-cache>
   <groups enabled="true"/>
</distributed-cache>
```

If you have control of the key class (you can alter the class definition, it's not part of an unmodifiable library), then we recommend using an intrinsic group. The intrinsic group is specified by adding the `@Group` annotation to a method. Let's take a look at an example:

```
class User {
   ...
   String office;
   ...

   public int hashCode() {
      // Defines the hash for the key, normally used to determine location
      ...
   }

   // Override the location by specifying a group
   // All keys in the same group end up with the same owners
   @Group
   public String getOffice() {
      return office;
   }
   }
}
```

The group method must return a `String`

If you don't have control over the key class, or the determination of the group is an orthogonal concern to the key class, we recommend using an extrinsic group. An extrinsic group is specified by implementing the `Grouper` interface.

```
public interface Grouper<T> {
    String computeGroup(T key, String group);

    Class<T> getKeyType();
}
```

If multiple `Grouper` classes are configured for the same key type, all of them will be called, receiving the value computed by the previous one. If the key class also has a `@Group` annotation, the first `Grouper` will receive the group computed by the annotated method. This allows you even greater control over the group when using an intrinsic group. Let's take a look at an example `Grouper` implementation:

```
public class KXGrouper implements Grouper<String> {

   // The pattern requires a String key, of length 2, where the first character is
   // "k" and the second character is a digit. We take that digit, and perform
   // modular arithmetic on it to assign it to group "0" or group "1".
   private static Pattern kPattern = Pattern.compile("(^k)(<a>\\d</a>)$");

   public String computeGroup(String key, String group) {
      Matcher matcher = kPattern.matcher(key);
      if (matcher.matches()) {
         String g = Integer.parseInt(matcher.group(2)) % 2 + "";
         return g;
      } else {
         return null;
      }
   }

   public Class<String> getKeyType() {
      return String.class;
   }
}
```

`Grouper` implementations must be registered explicitly in the cache configuration. If you are configuring Infinispan programmatically:

```
Configuration c = new ConfigurationBuilder()
   .clustering().hash().groups().enabled().addGrouper(new KXGrouper())
   .build();
```

Or, if you are using XML:

```
<distributed-cache>
   <groups enabled="true">
      <grouper class="com.acme.KXGrouper" />
   </groups>
</distributed-cache>
```

Advanced Interface

`AdvancedCache` has two group-specific methods:

getGroup(groupName)

Retrieves all keys in the cache that belong to a group.

removeGroup(groupName)

Removes all the keys in the cache that belong to a group.

Both methods iterate over the entire data container and store (if present), so they can be slow when a cache contains lots of small groups.

3.4. Scattered Caches

Scattered mode is very similar to Distribution Mode as it allows linear scaling of the cluster. It allows single node failure by maintaining two copies of the data (as Distribution Mode with numOwners=2). Unlike Distributed, the location of data is not fixed; while we use the same Consistent Hash algorithm to locate the primary owner, the backup copy is stored on the node that wrote the data last time. When the write originates on the primary owner, backup copy is stored on any other node (the exact location of this copy is not important).

This has the advantage of single Remote Procedure Call (RPC) for any write (Distribution Mode requires one or two RPCs), but reads have to always target the primary owner. That results in faster writes but possibly slower reads, and therefore this mode is more suitable for write-intensive applications.

Storing multiple backup copies also results in slightly higher memory consumption. In order to remove out-of-date backup copies, invalidation messages are broadcast in the cluster, which generates some overhead. This makes scattered mode less performant in very big clusters (this behaviour might be optimized in the future).

When a node crashes, the primary copy may be lost. Therefore, the cluster has to reconcile the backups and find out the last written backup copy. This process results in more network traffic during state transfer.

Since the writer of data is also a backup, even if we specify machine/rack/site ids on the transport level the cluster cannot be resilient to more than one failure on the same machine/rack/site.

Currently it is not possible to use scattered mode in transactional cache. Asynchronous replication is not supported either; use asynchronous Cache API instead. Functional commands are not implemented neither but these are expected to be added soon.

The cache is configured in a similar way as the other cache modes, here is an example of declarative configuration:

```
<scattered-cache name="scatteredCache" />
```

And this is how you can configure it programmatically:

```
Configuration c = new ConfigurationBuilder()
   .clustering().cacheMode(CacheMode.SCATTERED_SYNC)
   .build();
```

Scattered mode is not exposed in the server configuration as the server is usually accessed through the Hot Rod protocol. The protocol automatically selects primary owner for the writes and therefore the write (in distributed mode with two owner) requires single RPC inside the cluster, too.

Therefore, scattered cache would not bring the performance benefit.

3.5. Asynchronous Communication with Clustered Caches

3.5.1. Asynchronous Communications

All clustered cache modes can be configured to use asynchronous communications with the `mode="ASYNC"` attribute on the `<replicated-cache/>`, `<distributed-cache>`, or `<invalidation-cache/>` element.

With asynchronous communications, the originator node does not receive any acknowledgement from the other nodes about the status of the operation, so there is no way to check if it succeeded on other nodes.

We do not recommend asynchronous communications in general, as they can cause inconsistencies in the data, and the results are hard to reason about. Nevertheless, sometimes speed is more important than consistency, and the option is available for those cases.

3.5.2. Asynchronous API

The Asynchronous API allows you to use synchronous communications, but without blocking the user thread.

There is one caveat: The asynchronous operations do NOT preserve the program order. If a thread calls `cache.putAsync(k, v1); cache.putAsync(k, v2)`, the final value of `k` may be either `v1` or `v2`. The advantage over using asynchronous communications is that the final value can't be `v1` on one node and `v2` on another.

Prior to version 9.0, the asynchronous API was emulated by borrowing a thread from an internal thread pool and running a blocking operation on that thread.

3.5.3. Return Values in Asynchronous Communication

Because the `Cache` interface extends `java.util.Map`, write methods like `put(key, value)` and `remove(key)` return the previous value by default.

In some cases, the return value may not be correct:

1. When using `AdvancedCache.withFlags()` with `Flag.IGNORE_RETURN_VALUE`, `Flag.SKIP_REMOTE_LOOKUP`, or `Flag.SKIP_CACHE_LOAD`.
2. When the cache is configured with `unreliable-return-values="true"`.
3. When using asynchronous communications.
4. When there are multiple concurrent writes to the same key, and the cache topology changes. The topology change will make Infinispan retry the write operations, and a retried operation's return value is not reliable.

Transactional caches return the correct previous value in cases 3 and 4. However, transactional caches also have a gotcha: in distributed mode, the read-committed isolation level is implemented as repeatable-read. That means this example of "double-checked locking" won't work:

```
Cache cache = ...
TransactionManager tm = ...

tm.begin();
try {
   Integer v1 = cache.get(k);
   // Increment the value
   Integer v2 = cache.put(k, v1 + 1);
   if (Objects.equals(v1, v2) {
      // success
   } else {
      // retry
   }
} finally {
  tm.commit();
}
```

The correct way to implement this is to use `cache.getAdvancedCache().withFlags(Flag.FORCE_WRITE_LOCK).get(k)`.

In caches with optimistic locking, writes can also return stale previous values. Write skew checks can avoid stale previous values.

Chapter 4. Configuring Caches Declaratively

Infinispan declarative configuration.

4.1. Infinispan subsystem

The Infinispan subsystem configures the cache containers and caches.

The subsystem declaration is enclosed in the following XML element:

```
<subsystem xmlns="urn:infinispan:server:core:9.4" default-cache-container="clustered">
  ...
</subsystem>
```

4.1.1. Containers

The Infinispan subsystem can declare multiple containers. A container is declared as follows:

```
<cache-container name="clustered" default-cache="default">
  ...
</cache-container>
```

Infinispan does not provide an implicit default cache, but lets you name a cache as the default.

If you need to declare clustered caches (distributed, replicated, invalidation), you also need to specify the `<transport/>` element which references an existing JGroups transport. This is not needed if you only intend to have local caches only.

```
<transport executor="infinispan-transport" lock-timeout="60000" stack="udp" cluster=
"my-cluster-name"/>
```

4.1.2. Cache declarations

Now you can declare your caches. Please be aware that only the caches declared in the configuration will be available to the endpoints and that attempting to access an undefined cache is an illegal operation. Contrast this with the default Infinispan library behaviour where obtaining an undefined cache will implicitly create one using the default settings. The following are example declarations for all four available types of caches:

```
<local-cache name="default" start="EAGER">
  ...
</local-cache>

<replicated-cache name="replcache" mode="SYNC" remote-timeout="30000" start="EAGER">
  ...
</replicated-cache>

<invalidation-cache name="invcache" mode="SYNC" remote-timeout="30000" start="EAGER">
  ...
</invalidation-cache>

<distributed-cache name="distcache" mode="SYNC" segments="20" owners="2" remote-
timeout="30000" start="EAGER">
  ...
</distributed-cache>
```

4.2. Locking

To define the locking configuration for a cache, add the `<locking/>` element as follows:

```
<locking isolation="REPEATABLE_READ" acquire-timeout="30000" concurrency-level="1000"
striping="false"/>
```

The possible attributes for the locking element are:

- *isolation* sets the cache locking isolation level. Can be NONE, READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE. Defaults to REPEATABLE_READ
- *striping* if true, a pool of shared locks is maintained for all entries that need to be locked. Otherwise, a lock is created per entry in the cache. Lock striping helps control memory footprint but may reduce concurrency in the system.
- *acquire-timeout* maximum time to attempt a particular lock acquisition.
- *concurrency-level* concurrency level for lock containers. Adjust this value according to the number of concurrent threads interacting with Infinispan.
- *concurrent-updates* for non-transactional caches only: if set to true(default value) the cache keeps data consistent in the case of concurrent updates. For clustered caches this comes at the cost of an additional RPC, so if you don't expect your application to write data concurrently, disabling this flag increases performance.

4.3. Loaders and Stores

Loaders and stores can be defined in server mode in almost the same way as in embedded mode.

However, in server mode it is no longer necessary to define the `<persistence>⋯</persistence>` tag. Instead, a store's attributes are now defined on the store type element. For example, to configure

the H2 database with a distributed cache in domain mode we define the "default" cache as follows in our domain.xml configuration:

```
<subsystem xmlns="urn:infinispan:server:core:9.4">
  <cache-container name="clustered" default-cache="default" statistics="true">
    <transport lock-timeout="60000"/>
    <global-state/>
    <distributed-cache name="default">
      <string-keyed-jdbc-store datasource="java:jboss/datasources/ExampleDS" fetch-
state="true" shared="true">
        <string-keyed-table prefix="ISPN">
          <id-column name="id" type="VARCHAR"/>
          <data-column name="datum" type="BINARY"/>
          <timestamp-column name="version" type="BIGINT"/>
        </string-keyed-table>
        <write-behind modification-queue-size="20"/>
      </string-keyed-jdbc-store>
    </distributed-cache>
  </cache-container>
</subsystem>
```

Another important thing to note in this example, is that we use the "ExampleDS" datasource which is defined in the datasources subsystem in our domain.xml configuration as follows:

```
<subsystem xmlns="urn:jboss:domain:datasources:4.0">
  <datasources>
    <datasource jndi-name="java:jboss/datasources/ExampleDS" pool-name="ExampleDS"
enabled="true" use-java-context="true">
      <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-
1;DB_CLOSE_ON_EXIT=FALSE</connection-url>
      <driver>h2</driver>
      <security>
        <user-name>sa</user-name>
        <password>sa</password>
      </security>
    </datasource>
  </datasources>
</subsystem>
```

For additional examples of store configurations, please view the configuration templates in the default "domain.xml" file provided with in the server distribution at `./domain/configuration/domain.xml`.

4.4. State Transfer

To define the state transfer configuration for a distributed or replicated cache, add the `<state-transfer/>` element as follows:

```
<state-transfer enabled="true" timeout="240000" chunk-size="512" await-initial-
transfer="true" />
```

The possible attributes for the state-transfer element are:

- *enabled* if true, this will cause the cache to ask neighboring caches for state when it starts up, so the cache starts 'warm', although it will impact startup time. Defaults to true.
- *timeout* the maximum amount of time (ms) to wait for state from neighboring caches, before throwing an exception and aborting startup. Defaults to 240000 (4 minutes).
- *chunk-size* the number of cache entries to batch in each transfer. Defaults to 512.
- *await-initial-transfer* if true, this will cause the cache to wait for initial state transfer to complete before responding to requests. Defaults to true.

4.5. Declarative Cache Configuration

Declarative configuration comes in a form of XML document that adheres to a provided Infinispan configuration XML schema.

Every aspect of Infinispan that can be configured declaratively can also be configured programmatically. In fact, declarative configuration, behind the scenes, invokes the programmatic configuration API as the XML configuration file is being processed. One can even use a combination of these approaches. For example, you can read static XML configuration files and at runtime programmatically tune that same configuration. Or you can use a certain static configuration defined in XML as a starting point or template for defining additional configurations in runtime.

There are two main configuration abstractions in Infinispan: `global` and `cache`.

Global configuration

Global configuration defines global settings shared among all cache instances created by a single EmbeddedCacheManager. Shared resources like thread pools, serialization/marshalling settings, transport and network settings, JMX domains are all part of global configuration.

Cache configuration

Cache configuration is specific to the actual caching domain itself: it specifies eviction, locking, transaction, clustering, persistence etc. You can specify as many named cache configurations as you need. One of these caches can be indicated as the `default` cache, which is the cache returned by the `CacheManager.getCache()` API, whereas other named caches are retrieved via the `CacheManager.getCache(String name)` API.

Whenever they are specified, named caches inherit settings from the default cache while additional behavior can be specified or overridden. Infinispan also provides a very flexible inheritance mechanism, where you can define a hierarchy of configuration templates, allowing multiple caches to share the same settings, or overriding specific parameters as necessary.

One of the major goals of Infinispan is to aim for zero configuration. A simple XML configuration file containing nothing more than a single infinispan element is enough to get you started. The

configuration file listed below provides sensible defaults and is perfectly valid.

infinispan.xml

```
<infinispan />
```

However, that would only give you the most basic, local mode, non-clustered cache manager with no caches. Non-basic configurations are very likely to use customized global and default cache elements.

Declarative configuration is the most common approach to configuring Infinispan cache instances. In order to read XML configuration files one would typically construct an instance of DefaultCacheManager by pointing to an XML file containing Infinispan configuration. Once the configuration file is read you can obtain reference to the default cache instance.

```
EmbeddedCacheManager manager = new DefaultCacheManager("my-config-file.xml");
Cache defaultCache = manager.getCache();
```

or any other named instance specified in `my-config-file.xml`.

```
Cache someNamedCache = manager.getCache("someNamedCache");
```

The name of the default cache is defined in the `<cache-container>` element of the XML configuration file, and additional caches can be configured using the `<local-cache>`,`<distributed-cache>` ,`<invalidation-cache>` or `<replicated-cache>` elements.

The following example shows the simplest possible configuration for each of the cache types supported by Infinispan:

```
<infinispan>
   <cache-container default-cache="local">
      <transport cluster="mycluster"/>
      <local-cache name="local"/>
      <invalidation-cache name="invalidation" mode="SYNC"/>
      <replicated-cache name="repl-sync" mode="SYNC"/>
      <distributed-cache name="dist-sync" mode="SYNC"/>
   </cache-container>
</infinispan>
```

4.6. Cache configuration templates

As mentioned above, Infinispan supports the notion of *configuration templates*. These are full or partial configuration declarations which can be shared among multiple caches or as the basis for more complex configurations.

The following example shows how a configuration named `local-template` is used to define a cache

named `local`.

```
<infinispan>
   <cache-container default-cache="local">
      <!-- template configurations -->
      <local-cache-configuration name="local-template">
         <expiration interval="10000" lifespan="10" max-idle="10"/>
      </local-cache-configuration>

      <!-- cache definitions -->
      <local-cache name="local" configuration="local-template" />
   </cache-container>
</infinispan>
```

Templates can inherit from previously defined templates, augmenting and/or overriding some or all of the configuration elements:

```
<infinispan>
   <cache-container default-cache="local">
      <!-- template configurations -->
      <local-cache-configuration name="base-template">
         <expiration interval="10000" lifespan="10" max-idle="10"/>
      </local-cache-configuration>

      <local-cache-configuration name="extended-template" configuration="base-
template">
         <expiration lifespan="20"/>
         <memory>
            <object size="2000"/>
         </memory>
      </local-cache-configuration>

      <!-- cache definitions -->
      <local-cache name="local" configuration="base-template" />
      <local-cache name="local-bounded" configuration="extended-template" />
   </cache-container>
</infinispan>
```

In the above example, `base-template` defines a local cache with a specific *expiration* configuration. The `extended-template` configuration inherits from `base-template`, overriding just a single parameter of the *expiration* element (all other attributes are inherited) and adds a *memory* element. Finally, two caches are defined: `local` which uses the `base-template` configuration and `local-bounded` which uses the `extended-template` configuration.

Be aware that for multi-valued elements (such as `properties`) the inheritance is additive, i.e. the child configuration will be the result of merging the properties from the parent and its own.

4.7. Cache configuration wildcards

An alternative way to apply templates to caches is to use wildcards in the template name, e.g. `basecache*`. Any cache whose name matches the template wildcard will inherit that configuration.

```
<infinispan>
    <cache-container>
        <local-cache-configuration name="basecache*">
            <expiration interval="10500" lifespan="11" max-idle="11"/>
        </local-cache-configuration>
        <local-cache name="basecache-1"/>
        <local-cache name="basecache-2"/>
    </cache-container>
</infinispan>
```

Above, caches `basecache-1` and `basecache-2` will use the `basecache*` configuration. The configuration will also be applied when retrieving undefined caches programmatically.

If a cache name matches multiple wildcards, i.e. it is ambiguous, an exception will be thrown.

4.8. XInclude support

The configuration parser supports XInclude which means you can split your XML configuration across multiple files:

infinispan.xml

```
<infinispan xmlns:xi="http://www.w3.org/2001/XInclude">
    <cache-container>
        <local-cache name="cache-1"/>
        <xi:include href="included.xml" />
    </cache-container>
</infinispan>
```

included.xml

```
<local-cache name="cache-1"/>
```

the parser supports a minimal subset of the XInclude spec (no support for XPointer, fallback, text processing and content negotiation).

4.9. Declarative configuration reference

For more details on the declarative configuration schema, refer to the configuration reference.

If you are using XML editing tools for configuration writing you can use the provided Infinispan schema to assist you.

Chapter 5. Configuring Caches Programmatically

Infinispan programmatic configuration.

5.1. CacheManager and ConfigurationBuilder API

Programmatic Infinispan configuration is centered around the CacheManager and ConfigurationBuilder API. Although every single aspect of Infinispan configuration could be set programmatically, the most usual approach is to create a starting point in a form of XML configuration file and then in runtime, if needed, programmatically tune a specific configuration to suit the use case best.

```
EmbeddedCacheManager manager = new DefaultCacheManager("my-config-file.xml");
Cache defaultCache = manager.getCache();
```

Let's assume that a new synchronously replicated cache is to be configured programmatically. First, a fresh instance of Configuration object is created using ConfigurationBuilder helper object, and the cache mode is set to synchronous replication. Finally, the configuration is defined/registered with a manager.

```
Configuration c = new ConfigurationBuilder().clustering().cacheMode(CacheMode
.REPL_SYNC).build();

String newCacheName = "repl";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

The default cache configuration (or any other cache configuration) can be used as a starting point for creation of a new cache. For example, lets say that `infinispan-config-file.xml` specifies a replicated cache as a default and that a distributed cache is desired with a specific L1 lifespan while at the same time retaining all other aspects of a default cache. Therefore, the starting point would be to read an instance of a default Configuration object and use `ConfigurationBuilder` to construct and modify cache mode and L1 lifespan on a new `Configuration` object. As a final step the configuration is defined/registered with a manager.

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-file.xml");
Configuration dcc = manager.getDefaultCacheConfiguration();
Configuration c = new ConfigurationBuilder().read(dcc).clustering().cacheMode
(CacheMode.DIST_SYNC).l1().lifespan(60000L).build();

String newCacheName = "distributedWithL1";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

As long as the base configuration is the default named cache, the previous code works perfectly fine. However, other times the base configuration might be another named cache. So, how can new configurations be defined based on other defined caches? Take the previous example and imagine that instead of taking the default cache as base, a named cache called "replicatedCache" is used as base. The code would look something like this:

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-file.xml");
Configuration rc = manager.getCacheConfiguration("replicatedCache");
Configuration c = new ConfigurationBuilder().read(rc).clustering().cacheMode(
CacheMode.DIST_SYNC).l1().lifespan(60000L).build();

String newCacheName = "distributedWithL1";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

Refer to CacheManager , ConfigurationBuilder , Configuration , and GlobalConfiguration javadocs for more details.

5.2. ConfigurationBuilder Programmatic Configuration API

While the above paragraph shows how to combine declarative and programmatic configuration, starting from an XML configuration is completely optional. The ConfigurationBuilder fluent interface style allows for easier to write and more readable programmatic configuration. This approach can be used for both the global and the cache level configuration. GlobalConfiguration objects are constructed using GlobalConfigurationBuilder while Configuration objects are built using ConfigurationBuilder. Let's look at some examples on configuring both global and cache level options with this API:

5.2.1. Enabling JMX MBeans and statistics

Sometimes you might also want to enable collection of global JMX statistics at cache manager level or get information about the transport. To enable global JMX statistics simply do:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .globalJmxStatistics()
  .enable()
  .build();
```

Please note that by not enabling (or by explicitly disabling) global JMX statistics your are just turning off statistics collection. The corresponding MBean is still registered and can be used to manage the cache manager in general, but the statistics attributes do not return meaningful values.

Further options at the global JMX statistics level allows you to configure the cache manager name which comes handy when you have multiple cache managers running on the same system, or how to locate the JMX MBean Server:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .globalJmxStatistics()
    .cacheManagerName("SalesCacheManager")
    .mBeanServerLookup(new JBossMBeanServerLookup())
  .build();
```

5.2.2. Configuring thread pools

Some of the Infinispan features are powered by a group of the thread pool executors which can also be tweaked at this global level. For example:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
   .replicationQueueThreadPool()
     .threadPoolFactory(ScheduledThreadPoolExecutorFactory.create())
  .build();
```

You can not only configure global, cache manager level, options, but you can also configure cache level options such as the cluster mode:

```
Configuration config = new ConfigurationBuilder()
  .clustering()
    .cacheMode(CacheMode.DIST_SYNC)
    .sync()
    .l1().lifespan(25000L)
    .hash().numOwners(3)
  .build();
```

Or you can configure eviction and expiration settings:

```
Configuration config = new ConfigurationBuilder()
           .memory()
             .size(20000)
          .expiration()
             .wakeUpInterval(5000L)
             .maxIdle(120000L)
           .build();
```

5.2.3. Configuring transactions and locking

An application might also want to interact with an Infinispan cache within the boundaries of JTA and to do that you need to configure the transaction layer and optionally tweak the locking settings. When interacting with transactional caches, you might want to enable recovery to deal with transactions that finished with an heuristic outcome and if you do that, you will often want to enable JMX management and statistics gathering too:

```
Configuration config = new ConfigurationBuilder()
  .locking()
    .concurrencyLevel(10000).isolationLevel(IsolationLevel.REPEATABLE_READ)
    .lockAcquisitionTimeout(12000L).useLockStriping(false).writeSkewCheck(true)
    .versioning().enable().scheme(VersioningScheme.SIMPLE)
  .transaction()
    .transactionManagerLookup(new GenericTransactionManagerLookup())
    .recovery()
  .jmxStatistics()
  .build();
```

5.2.4. Configuring cache stores

Configuring Infinispan with chained cache stores is simple too:

```
Configuration config = new ConfigurationBuilder()
   .persistence().passivation(false)
   .addSingleFileStore().location("/tmp").async().enable()
   .preload(false).shared(false).threadPoolSize(20).build();
```

5.2.5. Advanced programmatic configuration

The fluent configuration can also be used to configure more advanced or exotic options, such as advanced externalizers:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .serialization()
    .addAdvancedExternalizer(998, new PersonExternalizer())
    .addAdvancedExternalizer(999, new AddressExternalizer())
  .build();
```

Or, add custom interceptors:

```
Configuration config = new ConfigurationBuilder()
  .customInterceptors().addInterceptor()
    .interceptor(new FirstInterceptor()).position(InterceptorConfiguration.Position
.FIRST)
    .interceptor(new LastInterceptor()).position(InterceptorConfiguration.Position
.LAST)
    .interceptor(new FixPositionInterceptor()).index(8)
    .interceptor(new AfterInterceptor()).after(NonTransactionalLockingInterceptor
.class)
    .interceptor(new BeforeInterceptor()).before(CallInterceptor.class)
  .build();
```

For information on the individual configuration options, please check the configuration guide.

Chapter 6. Setting Up Cluster Transport

Infinispan nodes rely on a transport layer to join and leave clusters as well as to replicate data across the network.

Infinispan uses JGroups technology to handle cluster transport. You configure cluster transport with JGroups stacks, which define properties for either UDP or TCP protocols.

6.1. Getting Started with Default Stacks

Use default JGroups stacks with recommended settings as a starting point for your cluster transport layer.

Default JGroups stacks are included in `infinispan-core.jar` and, as a result, are on the classpath.

Programmatic procedure

- Specify default JGroups stacks with the `addProperty()` method.

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder().transport()
        .defaultTransport()
        .clusterName("qa-cluster")
         // Use default JGroups stacks with the addProperty() method.
        .addProperty("configurationFile", "default-jgroups-tcp.xml")
        .machineId("qa-machine").rackId("qa-rack")
        .build();
```

Declarative procedure

- Specify default JGroups stacks with the `stack` attribute.

```
<infinispan>
  <cache-container default-cache="replicatedCache">
    <!-- Add default JGroups stacks to clustered caches. -->
    <transport stack="tcp" />
    ...
  </cache-container>
</infinispan>
```

Use the `cluster-stack` argument with the Infinispan server startup script.

```
$ bin/server.sh --cluster-stack=tcp
```

6.1.1. Default JGroups Stacks

File name	Stack name	Description
`default-jgroups-udp.xml`	`udp`	Uses UDP for transport and UDP multicast for discovery. Suitable for larger clusters (over 100 nodes) or if you are using replicated caches or invalidation mode. Minimises the number of open sockets.
`default-jgroups-tcp.xml`	`tcp`	Uses TCP for transport and UDP multicast for discovery. Suitable for smaller clusters (under 100 nodes) *only if* you are using distributed caches because TCP is more efficient than UDP as a point-to-point protocol.
`default-jgroups-ec2.xml`	`ec2`	Uses TCP for transport and `S3_PING` for discovery. Suitable for Amazon EC2 nodes where UDP multicast is not available.
`default-jgroups-kubernetes.xml`	`kubernetes`	Uses TCP for transport and `DNS_PING` for discovery. Suitable for Kubernetes and Red Hat OpenShift nodes where UDP multicast is not always available.
`default-jgroups-google.xml`	`google`	Uses TCP for transport and `GOOGLE_PING2` for discovery. Suitable for Google Cloud Platform nodes where UDP multicast is not available.
`default-jgroups-azure.xml`	`azure`	Uses TCP for transport and `AZURE_PING` for discovery. Suitable for Microsoft Azure nodes where UDP multicast is not available.

Next Steps

After you get up and running with the default JGroups stacks, use inheritance to combine, extend, remove, and replace stack properties. See Adjusting and Tuning JGroups Stacks.

6.1.2. Default JGroups Stacks

Infinispan uses the following JGroups `TCP` and `UDP` stacks by default:

```
<stack name="udp">
  <transport type="UDP" socket-binding="jgroups-udp"/>
  <protocol type="PING"/>
  <protocol type="MERGE3"/>
  <protocol type="FD_SOCK" socket-binding="jgroups-udp-fd"/>
  <protocol type="FD_ALL"/>
  <protocol type="VERIFY_SUSPECT"/>
  <protocol type="pbcast.NAKACK2"/>
  <protocol type="UNICAST3"/>
  <protocol type="pbcast.STABLE"/>
  <protocol type="pbcast.GMS"/>
  <protocol type="UFC_NB"/>
  <protocol type="MFC_NB"/>
  <protocol type="FRAG3"/>
</stack>
<stack name="tcp">
  <transport type="TCP" socket-binding="jgroups-tcp"/>
  <protocol type="MPING" socket-binding="jgroups-mping"/>
  <protocol type="MERGE3"/>
  <protocol type="FD_SOCK" socket-binding="jgroups-tcp-fd"/>
  <protocol type="FD_ALL"/>
  <protocol type="VERIFY_SUSPECT"/>
  <protocol type="pbcast.NAKACK2">
    <property name="use_mcast_xmit">false</property>
  </protocol>
  <protocol type="UNICAST3"/>
  <protocol type="pbcast.STABLE"/>
  <protocol type="pbcast.GMS"/>
  <protocol type="MFC_NB"/>
  <protocol type="FRAG3"/>
</stack>
```

To improve performance, Infinispan uses some values for properties other than the JGroups default values. You should examine the following files to review the JGroups configuration for Infinispan:

- Infinispan servers
  - `jgroups-defaults.xml`
  - `infinispan-jgroups.xml`
- Embedded Infinispan
  - `default-jgroups-tcp.xml`
  - `default-jgroups-udp.xml`

The default `TCP` stack uses the `MPING` protocol for discovery, which uses `UDP` multicast.

Reference

- JGroups Protocol

- JGroups Discovery Protocols

6.2. Using Inline JGroups Stacks

Use inline JGroups stack definitions to customize cluster transport for optimal network performance.

Use inheritance with inline JGroups stacks to tune and customize specific transport properties.

Procedure

- Embed your custom JGroups stack definitions in `infinispan.xml` as in the following example:

```
<infinispan>
  <!-- jgroups is the parent for stack declarations. -->
  <jgroups>
    <!-- Add JGroups stacks for Infinispan clustering. -->
    <stack name="prod">
      <TCP bind_port="7800" port_range="30" recv_buf_size="20000000" send_buf_size
="640000"/>
      <MPING bind_addr="127.0.0.1" break_on_coord_rsp="true"
             mcast_addr="${jgroups.mping.mcast_addr:228.2.4.6}"
             mcast_port="${jgroups.mping.mcast_port:43366}"
             num_discovery_runs="3"
             ip_ttl="${jgroups.udp.ip_ttl:2}"/>
      <MERGE3 />
      <FD_SOCK />
      <FD_ALL timeout="3000" interval="1000" timeout_check_interval="1000" />
      <VERIFY_SUSPECT timeout="1000" />
      <pbcast.NAKACK2 use_mcast_xmit="false" xmit_interval="100"
xmit_table_num_rows="50"
                      xmit_table_msgs_per_row="1024"
xmit_table_max_compaction_time="30000" />
      <UNICAST3 xmit_interval="100" xmit_table_num_rows="50"
xmit_table_msgs_per_row="1024"
                xmit_table_max_compaction_time="30000" />
      <pbcast.STABLE stability_delay="200" desired_avg_gossip="2000" max_bytes="1M"
/>
      <pbcast.GMS print_local_addr="false" join_timeout=
"${jgroups.join_timeout:2000}" />
      <UFC_NB max_credits="3m" min_threshold="0.40" />
      <MFC_NB max_credits="3m" min_threshold="0.40" />
      <FRAG3 />
    </stack>
  </jgroups>
  <cache-container default-cache="replicatedCache">
    <!-- Add JGroups stacks to clustered caches. -->
    <transport stack="prod" />
    ...
  </cache-container>
</infinispan>
```

Reference

Infinispan Configuration Schema

6.3. Adjusting and Tuning JGroups Stacks

Use inheritance to combine, extend, remove, and replace specific properties in the default JGroups stacks or custom configurations.

Procedure

1. Add a new JGroups stack declaration.
2. Name a parent stack with the `extends` attribute.
3. Modify transport properties with the `stack.combine` attribute.

For example, you want to evaluate using a Gossip router for cluster discovery using a `TCP` stack configuration named `prod`.

You can create a new stack named `gossip-prod` that inherits from `prod` and use `stack.combine` to change properties for the Gossip router configuration, as in the following example:

```
<jgroups>
  ...
  <!-- "gossip-prod" inherits properties from "prod" -->
  <stack name="gossip-prod" extends="prod">
    <!-- Use TCPGOSSIP discovery instead of MPING. -->
    <TCPGOSSIP initial_hosts="${jgroups.tunnel.gossip_router_hosts:localhost[12001]}"
             stack.combine="REPLACE" stack.position="MPING" />
    <!-- Remove FD_SOCK. -->
    <FD_SOCK stack.combine="REMOVE"/>
    <!-- Increase VERIFY_SUSPECT. -->
    <VERIFY_SUSPECT timeout="2000"/>
    <!-- Add SYM_ENCRYPT. -->
    <SYM_ENCRYPT sym_algorithm="AES"
                 key_store_name="defaultStore.keystore"
                 store_password="changeit"
                 alias="myKey" stack.combine="INSERT_AFTER" stack.position=
"pbcast.NAKACK2" />
  </stack>
  ...
</jgroups>
```

6.3.1. Stack Combine Attribute

`stack.combine` lets you override and modify inherited JGroups properties.

Value	Description
`COMBINE`	Overrides existing protocol attributes.
`REPLACE`	Replaces existing protocols that you identify with the `stack.position` attribute. If you do not specify `stack.position`, Infinispan defaults to the same protocol as the inherited configuration, which resets all non-specified attributes to the default values.
`INSERT_AFTER`	Inserts protocols after any protocols that you identify with the `stack.position` attribute.
`REMOVE`	Removes protocols from the inherited configuration.

6.4. Using JGroups Stacks in External Files

Use JGroups transport configuration from external files.

Infinispan looks for JGroups configuration files on your classpath first and then for absolute path names.

Programmatic procedure

- Specify your JGroups transport configuration with the `addProperty()` method.

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder().transport()
        .defaultTransport()
        .clusterName("prod-cluster")
         // Add custom JGroups stacks with the addProperty() method.
        .addProperty("configurationFile", "prod-jgroups-tcp.xml")
        .machineId("prod-machine").rackId("prod-rack")
        .build();
```

Declarative procedure

- Add your JGroups stack file and then configure the Infinispan cluster to use it.

```
<infinispan>
  <jgroups>
     <!-- Add custom JGroups stacks in external files. -->
     <stack-file name="prod-tcp" path="prod-jgroups-tcp.xml"/>
  </jgroups>
  <cache-container default-cache="replicatedCache">
    <!-- Add custom JGroups stacks to clustered caches. -->
    <transport stack="prod-tcp" />
    <replicated-cache name="replicatedCache"/>
  </cache-container>
  ...
</infinispan>
```

Reference

- GlobalConfigurationBuilder.transport()
- TransportConfigurationBuilder

6.5. Tuning JGroups Stacks with System Properties

Pass system properties to the JVM at startup to tune JGroups stacks.

For example, to change the `TCP` port and IP address do the following:

```
$ java -cp ... -Djgroups.tcp.port=1234 -Djgroups.tcp.address=192.0.2.0
```

6.5.1. System Properties for Default JGroups Stacks

`default-jgroups-udp.xml`

System Property	Description	Default Value	Required/Optional
`jgroups.udp.mcast_addr`	IP address for multicast, both discovery and inter-cluster communication. The IP address must be a valid "class D" address that is suitable for IP multicast.	`228.6.7.8`	Optional
`jgroups.udp.mcast_port`	Port for the multicast socket.	`46655`	Optional
`jgroups.udp.ip_ttl`	Specifies the time-to-live (TTL) for IP multicast packets. The value defines the number of network hops a packet can make before it is dropped.	2	Optional

`default-jgroups-tcp.xml`

System Property	Description	Default Value	Required/Optional
`jgroups.tcp.address`	IP address for TCP transport.	`127.0.0.1`	Optional
`jgroups.tcp.port`	Port for the TCP socket.	`7800`	Optional
`jgroups.udp.mcast_addr`	IP address for multicast discovery. The IP address must be a valid "class D" address that is suitable for IP multicast.	`228.6.7.8`	Optional
`jgroups.udp.mcast_port`	Port for the multicast socket.	`46655`	Optional
`jgroups.udp.ip_ttl`	Specifies the time-to-live (TTL) for IP multicast packets. The value defines the number of network hops a packet can make before it is dropped.	2	Optional

`default-jgroups-ec2.xml`

System Property	Description	Default Value	Required/Optional
`jgroups.tcp.address`	IP address for TCP transport.	`127.0.0.1`	Optional
`jgroups.tcp.port`	Port for the TCP socket.	`7800`	Optional
`jgroups.s3.access_key`	Amazon S3 access key for an S3 bucket.	No default value.	Optional
`jgroups.s3.secret_access_key`	Amazon S3 secret key used for an S3 bucket.	No default value.	Optional

System Property	Description	Default Value	Required/Optional
jgroups.s3.bucket	Name of the Amazon S3 bucket. The name must already exist and be unique.	No default value.	Optional

default-jgroups-kubernetes.xml

System Property	Description	Default Value	Required/Optional
jgroups.tcp.address	IP address for TCP transport.	eth0	Optional
jgroups.tcp.port	Port for the TCP socket.	7800	Optional

Reference

- JGroups System Properties
- JGroups Protocol List

6.6. Using Custom JChannels

Construct custom JGroups JChannels as in the following example:

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
JChannel jchannel = new JChannel();
// Configure the jchannel to your needs.
JGroupsTransport transport = new JGroupsTransport(jchannel);
global.transport().transport(transport);
new DefaultCacheManager(global.build());
```

Infinispan cannot use custom JChannels that are already connected.

Reference

JGroups JChannel

Chapter 7. Configuring Cluster Discovery

Running Infinispan on hosted services requires using discovery mechanisms that are adapted to network constraints that individual cloud providers impose. For instance, Amazon EC2 does not allow UDP multicast.

Infinispan can use the following cloud discovery mechanisms:

- Generic discovery protocols (`TCPPING` and `TCPGOSSIP`)
- JGroups PING protocols (`KUBE_PING` and `DNS_PING`)
- Cloud-specific PING protocols

Embedded Infinispan requires cloud provider dependencies.

7.1. TCPPING

`TCPPING` is a generic JGroups discovery mechanism that uses a static list of IP addresses for cluster members.

To use `TCPPING`, you must add the list of static IP addresses to the JGroups configuration file for each Infinispan node. However, the drawback to `TCPPING` is that it does not allow nodes to dynamically join Infinispan clusters.

TCPPING configuration example

```
<config>
      <TCP bind_port="7800" />
      <TCPPING timeout="3000"
           initial_hosts=
"${jgroups.tcpping.initial_hosts:localhost[7800],localhost[7801]}"
           port_range="1"
           num_initial_members="3"/>
...
...
</config>
```

Reference

JGroups TCPPING

7.2. Gossip Router

Gossip routers provide a centralized location on the network from which your Infinispan cluster can retrieve addresses of other nodes.

You inject the address (`IP:PORT`) of the Gossip router into Infinispan nodes as follows:

1. Pass the address as a system property to the JVM; for example,

`-DGossipRouterAddress="10.10.2.4[12001]"`.

2. Reference that system property in the JGroups configuration file.

Gossip router configuration example

```
<config>
    <TCP bind_port="7800" />
    <TCPGOSSIP timeout="3000" initial_hosts="${GossipRouterAddress}"
num_initial_members="3" />
...
...
</config>
```

Reference

JGroups Gossip Router

7.3. DNS_PING

JGroups `DNS_PING` queries DNS servers to discover Infinispan cluster members in Kubernetes environments such as OKD and Red Hat OpenShift.

DNS_PING configuration example

```
<stack name="dns-ping">
...
    <dns.DNS_PING
      dns_query="myservice.myproject.svc.cluster.local" />
...
</stack>
```

Reference

- JGroups DNS_PING
- DNS for Services and Pods (Kubernetes documentation for adding DNS entries)

7.4. KUBE_PING

JGroups `Kube_PING` uses a Kubernetes API to discover Infinispan cluster members in environments such as OKD and Red Hat OpenShift.

KUBE_PING configuration example

```
<config>
    <TCP bind_addr="${match-interface:eth.*}" />
    <kubernetes.KUBE_PING />
...
...
</config>
```

KUBE_PING configuration requirements

- Your `KUBE_PING` configuration must bind the JGroups stack to the `eth0` network interface. Docker containers use `eth0` for communication.
- `KUBE_PING` uses environment variables inside containers for configuration. The `KUBERNETES_NAMESPACE` environment variable must specify a valid namespace. You can either hardcode it or populate it via the Kubernetes Downward API.
- `KUBE_PING` requires additional privileges on Red Hat OpenShift. Assuming that `oc project -q` returns the current namespace and `default` is the service account name, you can run:

```
$ oc policy add-role-to-user view system:serviceaccount:$(oc project -q):default -n
$(oc project -q)
```

Reference

- JGroups Kube_PING
- Kubernetes Downward API
- Docker Networking

7.5. NATIVE_S3_PING

On Amazon Web Service (AWS), use the `S3_PING` protocol for discovery.

You can configure JGroups to use shared storage to exchange the details of Infinispan nodes. `NATIVE_S3_PING` allows Amazon S3 as the shared storage but requires both Amazon S3 and EC2 subscriptions.

NATIVE_S3_PING configuration example

```
<config>
    <TCP bind_port="7800" />
    <org.jgroups.aws.s3.NATIVE_S3_PING
            region_name="replace this with your region (e.g. eu-west-1)"
            bucket_name="replace this with your bucket name"
            bucket_prefix="replace this with a prefix to use for entries in the bucket
(optional)" />
</config>
```

NATIVE_S3_PING dependencies for embedded Infinispan

```
<dependency>
  <groupId>org.jgroups.aws.s3</groupId>
  <artifactId>native-s3-ping</artifactId>
  <!-- Replace ${version.jgroups.native_s3_ping} with the
  version of the native-s3-ping module you want to use. -->
  <version>${version.jgroups.native_s3_ping}</version>
</dependency>
```

7.6. JDBC_PING

JDBC_PING uses JDBC connections to shared databases, such as Amazon RDS on EC2, to store information about Infinispan nodes.

Reference

JDBC_PING Wiki

7.7. AZURE_PING

On Microsoft Azure, use a generic discovery protocol or AZURE_PING, which uses shared Azure Blob Storage to store discovery information.

AZURE_PING configuration example

```
<azure.AZURE_PING
    storage_account_name="replace this with your account name"
    storage_access_key="replace this with your access key"
    container="replace this with your container name"
/>
```

AZURE_PING dependencies for embedded Infinispan

```
<dependency>
  <groupId>org.jgroups.azure</groupId>
  <artifactId>jgroups-azure</artifactId>
  <!-- Replace ${version.jgroups.azure} with the
  version of the jgroups-azure module you want to use. -->
  <version>${version.jgroups.azure}</version>
</dependency>
```

7.8. GOOGLE2_PING

On Google Compute Engine (GCE), use a generic discovery protocol or GOOGLE2_PING, which uses Google Cloud Storage (GCS) to store information about the cluster members.

GOOGLE2_PING configuration example

```
<org.jgroups.protocols.google.GOOGLE_PING2 location="${jgroups.google.bucket_name}" />
```

GOOGLE2_PING dependencies for embedded Infinispan

```
<dependency>
  <groupId>org.jgroups.google</groupId>
  <artifactId>jgroups-google</artifactId>
  <!-- Replace ${version.jgroups.google} with the
  version of the jgroups-goole module you want to use. -->
  <version>${version.jgroups.google}</version>
</dependency>
```