Technical Overview

Infinispan 10.0

Table of Contents

Chapter 1. Introduction

Welcome to the official Infinispan documentation. This comprehensive document will guide you through every last detail of Infinispan. Because of this, it can be a poor starting point if you are new to Infinispan.

1.1. What is Infinispan ?

Infinispan is a distributed in-memory key/value data store with optional schema, available under the Apache License 2.0. It can be used both as an embedded Java library and as a language-independent service accessed remotely over a variety of protocols (Hot Rod, REST, Memcached and WebSockets). It offers advanced functionality such as transactions, events, querying and distributed processing as well as numerous integrations with frameworks such as the JCache API standard, CDI, Hibernate, WildFly, Spring Cache, Spring Session, Lucene, Spark and Hadoop.

1.2. Why use Infinispan ?

1.2.1. As a local cache

The primary use for Infinispan is to provide a fast in-memory cache of frequently accessed data. Suppose you have a slow data source (database, web service, text file, etc): you could load some or all of that data in memory so that it's just a memory access away from your code. Using Infinispan is better than using a simple ConcurrentHashMap, since it has additional useful features such as expiration and eviction.

1.2.2. As a clustered cache

If your data doesn't fit in a single node, or you want to invalidate entries across multiple instances of your application, Infinispan can scale horizontally to several hundred nodes.

1.2.3. As a clustering building block for your applications

If you need to make your application cluster-aware, integrate Infinispan and get access to features like topology change notifications, cluster communication and clustered execution.

1.2.4. As a remote cache

If you want to be able to scale your caching layer independently from your application, or you need to make your data available to different applications, possibly even using different languages / platforms, use Infinispan Server and its various clients.

1.2.5. As a data grid

Data you place in Infinispan doesn't have to be temporary: use Infinispan as your primary store and use its powerful features such as transactions, notifications, queries, distributed execution, distributed streams, analytics to process data quickly.

1.2.6. As a geographical backup for your data

Infinispan supports replication between clusters, allowing you to backup your data across geographically remote sites.

Chapter 2. Architectural Overview

This section contains a high level overview of Infinispan's internal architecture. This document is geared towards people with an interest in extending or enhancing Infinispan, or just curious about Infinispan's internals.

2.1. Cache hierarchy

Infinispan's Cache interface extends the JRE's ConcurrentMap interface which provides for a familiar and easy-to-use API.

```
public interface Cache<K, V> extends BasicCache<K, V> {
 ...
}

public interface BasicCache<K, V> extends ConcurrentMap<K, V> {
 ...
}
```

Caches are created by using a CacheContainer instance - either the EmbeddedCacheManager or a RemoteCacheManager. In addition to their capabilities as a factory for Caches, CacheContainers also act as a registry for looking up Caches.

EmbeddedCacheManagers create either clustered or standalone Caches that reside in the same JVM. RemoteCacheManagers, on the other hand, create RemoteCaches that connect to a remote cache tier via the Hot Rod protocol.

2.2. Commands

Internally, each and every cache operation is encapsulated by a command. These command objects represent the type of operation being performed, and also hold references to necessary parameters. The actual logic of a given command, for example a ReplaceCommand, is encapsulated in the command's perform() method. Very object-oriented and easy to test.

All of these commands implement the VisitableCommand interface which allow a Visitor (described in next section) to process them accordingly.

```
public class PutKeyValueCommand extends VisitableCommand {
  ...
}

public class GetKeyValueCommand extends VisitableCommand {
  ...
}

... etc ...
```

2.3. Visitors

Commands are processed by the various Visitors. The visitor interface, displayed below, exposes methods to visit each of the different types of commands in the system. This gives us a type-safe mechanism for adding behaviour to a call. Commands are processed by `Visitor`s. The visitor interface, displayed below, exposes methods to visit each of the different types of commands in the system. This gives us a type-safe mechanism for adding behaviour to a call.

```
public interface Vistor {
   Object visitPutKeyValueCommand(InvocationContext ctx, PutKeyValueCommand command)
throws Throwable;

   Object visitRemoveCommand(InvocationContext ctx, RemoveCommand command) throws
Throwable;

   Object visitReplaceCommand(InvocationContext ctx, ReplaceCommand command) throws
Throwable;

   Object visitClearCommand(InvocationContext ctx, ClearCommand command) throws
Throwable;

   Object visitPutMapCommand(InvocationContext ctx, PutMapCommand command) throws
Throwable;

   ... etc ...
}
```

An `AbstractVisitor` class in the `org.infinispan.commands` package is provided with no-op implementations of each of these methods. Real implementations then only need override the visitor methods for the commands that interest them, allowing for very concise, readable and testable visitor implementations.

2.4. Interceptors

Interceptors are special types of Visitors, which are capable of visiting commands, but also acts in a chain. A chain of interceptors all visit the command, one in turn, until all registered interceptors visit the command.

The class to note is the CommandInterceptor. This abstract class implements the interceptor pattern, and also implements Visitor. Infinispan's interceptors extend CommandInterceptor, and these add specific behaviour to specific commands, such as distribution across a network or writing through to disk.

There is also an experimental asynchronous interceptor which can be used. The interface used for asynchronous interceptors is AsyncInterceptor and a base implementation which should be used when a custom implementation is desired BaseCustomAsyncInterceptor. Note this class also implements the `Visitor` interface.

2.5. Putting it all together

So how does this all come together? Invocations on the cache cause the cache to first create an invocation context for the call. Invocation contexts contain, among other things, transactional characteristics of the call. The cache then creates a command for the call, making use of a command factory which initialises the command instance with parameters and references to other subsystems.

The cache then passes the invocation context and command to the InterceptorChain, which calls each and every registered interceptor in turn to visit the command, adding behaviour to the call. Finally, the command's perform() method is invoked and the return value, if any, is propagated back to the caller.

2.6. Subsystem Managers

The interceptors act as simple interception points and don't contain a lot of logic themselves. Most behavioural logic is encapsulated as managers in various subsystems, a small subset of which are:

2.6.1. DistributionManager

Manager that controls how entries are distributed across the cluster.

2.6.2. TransactionManager

Manager than handles transactions, usually supplied by a third party.

2.6.3. RpcManager

Manager that handles replicating commands between nodes in the cluster.

2.6.4. LockManager

Manager that handles locking keys when operations require them.

2.6.5. PersistenceManager

Manager that handles persisting data to any configured cache stores.

2.6.6. DataContainer

Container that holds the actual in memory entries.

2.6.7. Configuration

A component detailing all of the configuration in this cache.

2.7. ComponentRegistry

A registry where the various managers above and components are created and stored for use in the cache. All of the other managers and crucial components are accessible through the registry.

The registry itself is a lightweight dependency injection framework, allowing components and managers to reference and initialise one another. Here is an example of a component declaring a dependency on a DataContainer and a Configuration, and a DataContainerFactory declaring its ability to construct DataContainers on the fly.

```
@Inject
public void injectDependencies(DataContainer container, Configuration configuration) {
   this.container = container;
   this.configuration = configuration;
}

@DefaultFactoryFor
public class DataContainerFactory extends AbstractNamedCacheComponentFactory {
```

Components registered with the ComponentRegistry may also have a lifecycle, and methods annotated with @Start or @Stop will be invoked before and after they are used by the component registry.

```
@Start
public void init() {
   useWriteSkewCheck = configuration.locking().writeSkewCheck();
}

@Stop(priority=20)
public void stop() {
   notifier.removeListener(listener);
   executor.shutdownNow();
}
```

In the example above, the optional priority parameter to @Stop is used to indicate the order in which the component is stopped, in relation to other components. This follows a Unix Sys-V style ordering, where smaller priority methods are called before higher priority ones. The default priority, if not specified, is 10.

Welcome to Infinispan's Frequently Asked Questions document. We hope you find the answers to your queries here, however if you don't, we encourage you to connect with the Infinispan community and ask any questions you may have on the Infinispan User Forums.

2.8. Project questions

2.8.1. What is Infinispan?

Infinispan is an open source data grid platform. It exposes a JSR-107 compatible `Cache` interface (which in turn extends `java.util.Map`) in which you can store objects. While Infinispan can be run in local mode, its real value is in distributed mode where caches cluster together and expose a large memory heap. Distributed mode is more powerful than simple replication since each data entry is spread out only to a fixed number of replicas thus providing resilience to server failures as well as scalability since the work done to store each entry is constant in relation to a cluster size.

So, why would you use it? Infinispan offers:

- *Massive heap* and *high availability* - If you have 100 blade servers, and each node has 2GB of space to dedicate to a replicated cache, you end up with 2 GB of total data. Every server is just a copy. On the other hand, with a distributed grid - assuming you want 1 copy per data item - you get a 100 GB memory backed virtual heap that is efficiently accessible from anywhere in the grid. If a server fails, the grid simply creates new copies of the lost data, and puts them on other servers. Applications looking for ultimate performance are no longer forced to delegate the majority of their data lookups to a large single database server - a bottleneck that exists in over 80% of enterprise applications!
- *Scalability* - Since data is evenly distributed there is essentially no major limit to the size of the grid, except group communication on the network - which is minimised to just discovery of new nodes. All data access patterns use peer-to-peer communication where nodes directly speak to each other, which scales very well. Infinispan does not require entire infrastructure shutdown to allow scaling up or down. Simply add/remove machines to your cluster without incurring any down-time.
- *Data distribution* - Infinispan uses consistent hash algorithm to determine where keys should be located in the cluster. Consistent hashing allows for cheap, fast and above all, deterministic location of keys with no need for further metadata or network traffic. The goal of data distribution is to maintain enough copies of state in the cluster so it can be durable and fault tolerant, but not too many copies to prevent Infinispan from being scalable.
- *Persistence* - Infinispan exposes a CacheStore interface, and several high-performance implementations - including JDBC cache stores, lesystem-based cache stores, Amazon S3 cache stores, etc. CacheStores can be used for "warm starts", or simply to ensure data in the grid survives complete grid restarts, or even to over ow to disk if you really do run out of memory.
- *Language bindings* (PHP, Python, Ruby, C, etc.) - Infinispan offers support for both the popular memcached protocol - with existing clients for almost every popular programming language - as well as an optimised Infinispan-specific protocol called Hot Rod. This means that Infinispan is not just useful to Java. Any major website or application that wants to take advantage of a fast data grid will be able to do so.
- *Management* - When you start thinking about running a grid on several hundred servers, management is no longer an extra, it becomes a necessity. Since version 8.0, Infinispan bundles a management console.
- *Support for Compute Grids* - Infinispan 5 adds the ability to pass a Runnable around the grid. This allows you to push complex processing towards the server where data is local, and pull back results using a Future. This map/reduce style paradigm is common in applications where a large amount of data is needed to compute relatively small results.

Also see this page on the Infinispan website.

2.8.2. What would I use Infinispan for?

Most people use Infinispan for one of two reasons. Firstly, as a distributed cache. Putting Infinispan in front of your database, disk-based NoSQL store or any part of your system that is a bottleneck can greatly help improve performance. Often, a simple cache isn't enough - for example if your application is clustered and cache coherency is important to data consistency. A distributed cache can greatly help here.

The other major use-case is as a NoSQL data store. In addition to being in memory, Infinispan can also persist data to a more permanent store. We call this a cache store. Cache stores are pluggable, you can easily write your own, and many already exist for you to use.

A less common use case is adding clusterability and high availability to frameworks. Since Infinispan exposes a distributed data structure, frameworks and libraries that also need to be clustered can easily achieve this by embedding Infinispan and delegating all state management to Infinispan. This way, any framework can easily be clustered by letting Infinispan do all the heavy lifting.

2.8.3. How is Infinispan related to JBoss Cache?

Certain design ideas and indeed some code have been borrowed from JBoss Cache 3.x, however JBoss Cache is in no way a dependency. Infinispan is a complete, separate and standalone project. Some may consider this a fork, but the people behind Infinispan and JBoss Cache see it as an evolution, since all future effort will be on Infinispan and not JBoss Cache.

2.8.4. What version of Java does Infinispan need to run? Does Infinispan need an application server to run?

All that is needed is a Java 8 compatible JVM. An application server is *not* a requirement.

2.8.5. Will there be a POJO Cache replacement in Infinispan?

Yes, and this is called Hibernate OGM .

2.8.6. How come Infinispan's first release is 4.0.0? This sounds weird!

We didn't want to release Infinispan as a 1.0, as in all fairness it is not a virgin codebase. A lot of the code, designs and ideas in Infinispan are from JBoss Cache, and has been tried and tested, proven in high stress environments. Infinispan should thus be viewed as a mature and stable platform and not a new, experimental one.

2.8.7. How is this related to JSR 107, the JCACHE specification?

Infinispan core engineers are on the JSR 107 expert group and starting with version 7.0.0, Infinispan provides a certified compatible implementation of version 1.0.0 of the specification.

2.8.8. Can I use Infinispan with Hibernate?

Yes, you can combine one or more of these integrations in the same application:

- *Using Infinispan as a database replacement*: using Hibernate OGM you can replace the RDBMS and store your entities and relations directly in Infinispan, interacting with it through the well known JPA 2.1 interface, with some limitations in the query capabilities. Hibernate OGM also automates mapping, encoding and decoding of JPA entities to Protobuf. For more details see Hibernate OGM .
- *Caching database access*: Hibernate can cache frequently loaded entities and queries in Infinispan, taking advantage of state of the art eviction algorithms, and clustering if needed but it provides a good performance boost in non-clustered deployments too.
- *Storing Lucene indexes*: When using Hibernate Search to provide full-text capabilities to your Hibernate/JPA enabled application, you need to store an Apache Lucene index separately from the database. You can store the index in Infinispan: this is ideal for clustered applications since it's otherwise tricky to share the index with correct locking on shared file systems, but is an interesting option for non-clustered deployments as well as it can combine the benefits of in-memory performance with reliability and write-through to any CacheStore supported by Infinispan.
- *Using full-text queries on Infinispan*: If you liked the powerful full-text and data mining capabilities of Hibernate Search, but don't need JPA or a database, you can use the indexing and query engine only: the Infinispan Query module reuses Hibernate Search internally, depending on some Hibernate libraries but exposing the Search capabilities only.
- A *combination of multiple such integrations*: you can use Hibernate OGM as an interface to perform CRUD operations on some Infinispan caches configured for resiliency, while also activating Hibernate 2nd level caching using some different caches configured for high performance read mostly access, and also use Hibernate Search to index your domain model while storing the indexes in Infinispan itself.

2.9. Technical questions

2.9.1. General questions

What APIs does Infinispan offer?

Infinispan's primary API - `org.infinispan.Cache` - extends `java.util.concurrent.ConcurrentMap` and closely resembles `javax.cache.Cache` from JSR 107. This is the most performant API to use, and should be used for all new projects.

Which JVMs (JDKs) does Infinispan work with?

Infinispan is developed and primarily tested against Oracle Java SE 8. It should work with most Java SE 8 implementations, including those from IBM, HP, Apple, Oracle, and OpenJDK.

Does Infinispan store data by value or by reference?

By default, Infinispan stores data by reference. So once clients store some data, clients can still

modify entries via original object references. This means that since client references are valid, clients can make changes to entries in the cache using those references, but these modifications are only local and you still need to call one of the cache's put/replace... methods in order for changes to replicate.

Obviously, allowing clients to modify cache contents directly, without any cache invocation, has some risks and that's why Infinispan offers the possibility to store data by value instead. The way store-by-value is enabled is by enabling Infinispan to store data in binary format and forcing it to do these binary transformations eagerly.

The reason Infinispan stores data by-reference instead of by-value is performance. Storing data by reference is quicker than doing it by value because it does not have the penalty of having to transform keys and values into their binary format.

Can I use Infinispan with Groovy? What about Jython, Clojure, JRuby or Scala etc.?

While we haven't extensively tested Infinispan on anything other than Java, there is no reason why it cannot be used in any other environment that sits atop a JVM. We encourage you to try, and we'd love to hear your experiences on using Infinispan from other JVM languages.

2.9.2. Cache Loader and Cache Store questions

Are modifications to asynchronous cache stores coalesced or aggregated?

Modifications are coalesced or aggregated for the interval that the modification processor thread is currently applying. This means that while changes are being queued, if multiple modifications are made to the same key, only the key's last state will be applied, hence reducing the number of calls to the cache store.

What does the passivation flag do?

Passivation is a mode of storing entries in the cache store *only when* they are evicted from memory. The benefit of this approach is to prevent a lot of expensive writes to the cache store if an entry is hot (frequently used) and hence *not* evicted from memory. The reverse process, known as *activation*, occurs when a thread attempts to access an entry which is *not* in memory but is in the store (i.e., a *passivated* entry). Activation involves loading the entry into memory, and then *removing* it from the cache store. With passivation enabled, the cache uses the cache store as an overflow tank, akin to swapping memory pages to disk in virtual memory implementations in operating systems.

If passivation is disabled, the cache store behaves as a write-through (or write-behind if asynchronous) cache, where all entries in memory are also maintained in the cache store. The effect of this is that the cache store will always contain a superset of what is in memory.

What if I get IOException "Unsupported protocol version 48" with JdbcStringBasedCacheStore?

You have probably set your data column type to `VARCHAR`, `CLOB` or something similar, but it should be `BLOB/VARBINARY`. Even though it's called `JdbcStringBasedCacheStore`, only the keys are required to be strings; the values can be anything, so they need to be stored in a binary column. See the

setDataColumnType javadoc for more details.

Is there any way I can boost cache store's performance?

If, for put operations, you don't need the previous values existing in the cache/store then the following optimisation can be made:

```
cache.getAdvancedCache().withFlags(Flag.SKIP_CACHE_LOAD).put(key, value);
```

Note that in this case the value returned by `cache.put()` is not reliable. This optimization skips a cache store read and can have very significant performance improvement effects.

2.9.3. Locking and Transaction questions

Does Infinispan support distributed eager locking?

Yes it does. By default, transactions are optimistic, and locks are only acquired during the prepare phase. However, Infinispan can be configured to lock cache keys eagerly, by using the pessimistic locking mode:

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.transaction().lockingMode(LockingMode.PESSIMISTIC);
```

With pessimistic locking, Infinispan will implicitly acquire locks when a transaction modifies one or more keys:

```
tm.begin()
cache.put(K,V)    // acquire cluster-wide lock on K
cache.put(K2,V2)  // acquire cluster-wide lock on K2
cache.put(K,V5)   // no-op, we already own cluster wide lock for K
tm.commit()       // releases locks
```

How does Infinispan support explicit eager locking?

When the cache is configured with pessimistic locking, the `lock(K…)` method allows cache users to explicitly lock set of cache keys eagerly during a transaction. Lock call attempts to lock specified cache keys on the proper lock owners and it either succeeds or fails. All locks are released during commit or rollback phase.

```
tm.begin()
cache.getAdvancedCache().lock(K)  // acquire cluster-wide lock on K
cache.put(K,V5)                   // guaranteed to succeed
tm.commit()                       // releases locks
```

What isolation levels does Infinispan support?

Infinispan only supports the isolation levels **READ_COMMITTED** and **REPEATABLE_READ**. Note that exact definition of these levels may differ from traditional database definitions.

The default isolation mode is **READ_COMMITTED**. We consider **READ_COMMITTED** to be good enough for most applications and hence its use as a default.

When using Atomikos transaction manager, distributed caches are not distributing data, what is the problem?

For efficiency reasons, Atomikos transaction manager commits transactions in a separate thread to the thread making the cache operations and until 4.2.1.CR1, Infinispan had problems with this type of scenarios and resulted on distributed caches not sending data to other nodes (see ISPN-927 for more details). Please note that replicated, invalidated or local caches would work fine. It's only distributed caches that would suffer this problem.

There're two ways to get around this issue, either:

1. Upgrade to Infinispan 4.2.1.CR2 or higher where the issue has been fixed.
2. If using Infinispan 4.2.1.CR1 or earlier, configure Atomikos so that `com.atomikos.icatch.threaded_2pc` is set to false . This results in commits happening in the same thread that made the cache operations.

2.9.4. Eviction and Expiration questions

Expiration does not work, what is the problem?

Multiple cache operations such as `put()` can take a lifespan as parameter which defines the time when the entry should be expired. If you have no eviction configured and and you let this time expire, it can look as Infinispan has not removed the entry. For example, the JMX stats such as number of entries might not updated or the persistent store associated with Infinispan might still contain the entry. To understand what's happening, it's important to note that Infinispan has marked the entry as expired but has not actually removed it. Removal of *expired* entries happens in one of 2 ways:

1. You try and do a get() or containsKey() for that entry. The entry is then detected as expired and is removed.
2. You have enabled eviction and an eviction thread wakes up periodically and purges expired entries.

If you have not enabled (2), or your eviction thread wakeup interval is large and you probe jconsole before the eviction thread kicks in, you will still see the expired entry. You can be assured that if you tried to *retrieve* the entry via a get() or containsKey() though, you won't see the entry (and the entry will be removed).

2.9.5. Cache Manager questions

Can I create caches using different cache modes using the same cache manager?

Yes. You can create caches using different cache modes, both synchronous and asynchronous, using the same cache manager.

Can transactions span different Cache instances from the same cache manager?

Yes. Each cache behaves as a separate, standalone JTA resource. Internally though, components may be shared as an optimization but this in no way affects how the caches interact with a JTA manager.

How does multi-tenancy work?

Multi-tenancy is achieved by namespacing. A single Infinispan cluster can have several named caches (attached to the same CacheManager), and different named caches can have duplicate keys. So this is, in effect, multi-tenancy for your key/value store.

Infinispan allows me to create several Caches from a single CacheManager. Are there any reasons to create separate CacheManagers?

As far as possible, internal components are shared between Cache instances. Notably, RPC and networking components are shared. If you need caches that have different network characteristics - such as one cache using TCP while another uses UDP - we recommend you create these using different cache managers.

2.9.6. Cache Mode questions

What is the difference between a replicated cache and a distributed cache?

Distribution is a new cache mode in Infinispan, in addition to replication and invalidation. In a replicated cache all nodes in a cluster hold all keys i.e. if a key exists on one node, it will also exist on *all* other nodes. In a distributed cache, a number of copies are maintained to provide redundancy and fault tolerance, however this is typically far fewer than the number of nodes in the cluster. A distributed cache provides a far greater degree of scalability than a replicated cache.

A distributed cache is also able to transparently locate keys across a cluster, and provides an L1 cache for fast local read access of state that is stored remotely.

Does DIST support both synchronous and asynchronous communications?

Officially, no. And unofficially, yes. Here's the logic. For certain public API methods to have meaningful return values (i.e., to stick to the interface contracts), if you are using DIST , synchronized communications are necessary. For example, you have 3 caches in a cluster, A, B and C. Key K maps to A and B. On C, you perform an operation that requires a return value e.g., Cache.remove(K) . For this to work, the call needs to be forwarded to A and B *synchronously*, and would have to wait for the result from either A or B to return to the caller. If communications were asynchronous, the return values cannot be guaranteed to be useful - even though the operation would behave as expected.

Now unofficially, we will add a configuration option to allow you to set your cache mode to DIST

and use asynchronous communications, but this would be an additional configuration option (perhaps something like break_api_contracts) so that users are aware of what they are getting into.

I notice that when using DIST, the cache does a remote get before a write command. Why is this?

Certain methods, such as Cache.put() , are supposed to return the previous value associated with the specified key according to the java.util.Map contract. If this is performed on an instance that does *not* own the key in question and the key is not in L1 cache, the only way to reliably provide this return value is to do a remote GET before the put. This GET is *always* sync (regardless of whether the cache is configured to be sync or async) since we need to wait for that return value.

Isn't that expensive? How can I optimize this away?

It isn't as expensive as it sounds. A remote GET, although sync, will *not* wait for all responses. It will accept the first valid response and move on, thus making its performance has no relation to cluster size.

If you feel your code has no need for these return values, then this can be disabled completely (by specifying the `<unsafe unreliableReturnValues="true" />` configuration element for a cache-wide setting or the `Flag.SKIP_REMOTE_LOOKUP` for a per-invocation setting). Note that while this will *not* impair cache operations and accurate functioning of all public methods is still maintained. However, it *will* break the java.util.Map interface contract by providing unreliable and inaccurate return values to certain methods, so you would need to be certain that your code does not use these return values for anything useful.

I use a clustered cache. I want the guarantees of synchronous replication with the parallelism of asynchronous replication. What can I do?

Infinispan offers a new async API to provide just this. These async methods return Future which can be queried, causing the thread to block till you get a confirmation that any network calls succeeded. You can read more about it .

What is the L1 cache?

An L1 cache (disabled by default) only exists if you set your cache mode to distribution. An L1 cache prevents unnecessary remote fetching of entries mapped to remote caches by storing them locally for a short time after the first time they are accessed. By default, entries in L1 have a lifespan of 60,000 milliseconds (though you can configure how long L1 entries are cached for). L1 entries are also invalidated when the entry is changed elsewhere in the cluster so you are sure you don't have stale entries cached in L1. Caches with L1 enabled will consult the L1 cache before fetching an entry from a remote cache.

What consistency guarantees do I have with different Asynchronous processing settings ?

There are 3 main configuration settings (modes of usage) that affect the behaviour of Infinispan in terms of Asynchronous processing, summarized in the following table:

Config / Mode of usage	Description
API	Usage of Asynchronous API, i.e. methods of the Cache interface like e.g. putAsync(key, val)
Replication	Configuring a clustered cache to replicate data asychronously. In Infinispan XML configuration this is done by using <sync> or <async> sub-elements under <clustering> element.

Switching to asynchronous mode in each of these areas causes loss of some consistency guarantees. The known problems are summarised here:

API	Replication	Marshalling	Consistency problems
Sync	Sync	Sync	
Sync	*Async*	Sync	*1* - Cache entry is replicated with a delay or not at all in case of network error. *2* - Node where the operation originated won't be notified about errors that happened on network or on the receiving side.
Sync	*Async*	*Async*	*1, 2 3* - Calling order of sync API method might not be preserved – depends on which operation finishes marshalling first in the asyncExecutor *4* - Replication of put operation can be applied on different nodes in different order – this may result in inconsistent values
Async	Sync	Sync	*3*
Async	*Async*	Sync	*1, 2, 3*
Async	*Async*	*Async*	*1, 2, 3, 4*

Grouping API vs Key Affinity Service

The key affinity (for keys generated with the Key Affinity Service) might be lost during topology changes. E.g. if k1 maps to node N1 and another node is added to the system, k1 can me migrated to N2 (affinity is lost). With grouping API you have the guarantee that the same node (you don't know/control which node) hosts all the data from the same group even after topology changes.

2.9.7. Listener questions

In a cache entry modified listener, can the modified value be retrieved via Cache.get() when isPre=false?

No, it cannot. Use `CacheEntryModifiedEvent.getValue()` to retrieve the value of the entry that was modified.

When annotating a method with CacheEntryCreated, how do I retrieve the value of the cache entry added?

Use `CacheEntryCreatedEvent.getValue()` to retrieve the value of the entry.

What is the difference between classes in `org.infinispan.notifications.cachelistener.filter` vs `org.infinispan.filter`?

Inside these packages you'll find classes that facilitate filtering and data conversion. The difference is that classes in `org.infinispan.filter` are used for filtering and conversion in multiple areas, such as cache loaders, entry iterators,...etc, whereas classes in `org.infinispan.notifications.cachelistener.filter` are purely used for listener event filtering, and provide more information than similarly named classes in `org.infinispan.filter`. More specifically, remote listener event filtering and conversion require `CacheEventFilter` and `CacheEventConverter` instances located in `org.infinispan.notifications.cachelistener.filter` package to be used.

2.9.8. IaaS/Cloud Infrastructure questions

How do you make Infinispan send replication traffic over a specific network when you don't know the IP address?

Some cloud providers charge you less for traffic over internal IP addresses compared to public IP addresses, in fact, some cloud providers do not even charge a thing for traffic over the internal network (i.e. GoGrid). In these circumstances, it's really advantageous to configure Infinispan in such way that replication traffic is sent via the internal network. The problem though is that quite often you don't know which internal IP address you'll be assigned (unless you use elastic IPs and dyndns.org), so how do you configure Infinispan to cope with those situations?

JGroups, which is the underlying group communication library to interconnect Infinispan instances, has come up with a way to enable users to bind to a type of address rather than to a specific IP address. So now you can configure `bind_addr` property in JGroups configuration file, or the `-Djgroups.bind_addr` system property to a keyword rather than a dotted decimal or symbolic IP address:

- GLOBAL : pick a public IP address. You want to avoid this for replication traffic
- SITE_LOCAL : use a private IP address, e.g. 192.168.x.x. This avoids charges for bandwidth from GoGrid, for example
- LINK_LOCAL : use a 169.x.x.x, 254.0.0.0 address. I've never used this, but this would be for traffic only within 1 box
- NON_LOOPBACK : use the first address found on an interface (which is up), which is not a 127.x.x.x address

2.9.9. Demo questions

When using the GUI Demo, I've just put an entry in the cache with lifespan of -1. Why do I see it as having a lifespan of 60,000?

This is probably a L1 caching event. When you put an entry in the cache, the entry is mapped to specific nodes in a cluster using a consistent hashing algorithm. This means that key K could map on to caches A and B (or however many owners you have configured). If you happen to have done the cache.put(K, V) on cache C , however, K still maps to A and B (and will be added to caches A and B with their proper lifespans), but it will also be put in cache C's L1 cache.

2.9.10. Logging questions

How can I enable logging?

By default Infinispan uses JBoss Logging 3.0 as logging framework. JBoss Logging acts as a delegator to either JBoss Log Manager, Apache Log4j, Slf4j or JDK Logging. The way it chooses which logging provider to delegate to is by:

1. checking whether the JBoss Log Manager is configured (e.g. Infinispan is running in JBoss Application Server 7) and if it is, using it
2. otherwise, checking if Apache Log4j is in the classpath (JBoss Logging checks if the classes org.apache.log4j.LogManager and org.apache.log4j.Hierarchy are available) and if it is, using it
3. otherwise, checking if LogBack in the classpath (JBoss Logging checks if the class ch.qos.logback.classic.Logger is available) and if it is, using it
4. finally, if none of the above are available, using JDK logging

You can use this log4j2.xml as base for any Infinispan related logging, and you can pass it to your system via system parameter (e.g., `-Dlog4j.configurationFile=file:/path/to/log4j2.xml`).

2.9.11. Third Party Container questions

Can I use Infinispan on Google App Engine for Java?

Not at this moment. Due to GAE/J restricting classes that can be loaded, and restrictions around use of threads, Infinispan will not work on GAE/J. However, we do plan to fix this - if you wish to track the progress of Infinispan on GAE/J, have a look at ISPN-57 .

When running on Glassfish or Apache, creating a cache throws an exception saying "Unable to construct a GlobalComponentRegistry", what is it wrong?

It appears that this happens due to some classloading issue. A workaround that is know to work is to call the following before creating the cache manager or container:

```
Thread.currentThread().setContextClassLoader(this.getClass().getClassLoader());
```

2.9.12. Marshalling and Unmarshalling

Best practices implementing java.io.Externalizable

If you decide to implement Externalizable interface, please make sure that the readExternal() method is thread safe, otherwise you run the risk of potential getting corrupted data and OutOfMemoryException , as seen in this forum post .

Do Externalizer implementations need to access internal Externalizer implementations?

No, they don't. Here's an example of what should not be done:

```
public static class ABCMarshallingExternalizer implements AdvancedExternalizer
<ABCMarshalling> {
   @Override
   public void writeObject(ObjectOutput output, ABCMarshalling object) throws
IOException {
      MapExternalizer ma = new MapExternalizer();
      ma.writeObject(output, object.getMap());
   }

   @Override
   public ABCMarshalling readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
      ABCMarshalling hi = new ABCMarshalling();
      MapExternalizer ma = new MapExternalizer();
      hi.setMap((ConcurrentHashMap<Long, Long>) ma.readObject(input));
      return hi;
   }

   ...
}
```

End user externalizers should not need to fiddle with Infinispan internal externalizer classes. Instead, this code should have been written as:

```
public static class ABCMarshallingExternalizer implements AdvancedExternalizer
<ABCMarshalling> {
   @Override
   public void writeObject(ObjectOutput output, ABCMarshalling object) throws
IOException {
      output.writeObject(object.getMap());
   }

   @Override
   public ABCMarshalling readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
      ABCMarshalling hi = new ABCMarshalling();
      hi.setMap((ConcurrentHashMap<Long, Long>) input.readObject());
      return hi;
   }

   ...
}
```

During state transfer, the state receiver logs an EOFException when applying state saying "Read past end of file". Should I worry about this?

It depends on whether the state provider encountered an error or not when generating the state. For example, sometimes the state provider might already be providing state to another node, so when the node requests the state, the state generator might log:

```
ERROR [org.infinispan.remoting.transport.jgroups.JGroupsTransport]
(STREAMING_STATE_TRANSFER-sender-1,{brandname}-Cluster,NodeJ-2368:) Caught while
responding to state transfer request
org.infinispan.statetransfer.StateTransferException:
java.util.concurrent.TimeoutException: Could not obtain exclusive processing lock
     at
org.infinispan.statetransfer.StateTransferManagerImpl.generateState(StateTransferManag
erImpl.java:175)
     at
org.infinispan.remoting.InboundInvocationHandlerImpl.generateState(InboundInvocationHa
ndlerImpl.java:119)
     at
org.infinispan.remoting.transport.jgroups.JGroupsTransport.getState(JGroupsTransport.j
ava:586)
     at
org.jgroups.blocks.MessageDispatcher$ProtocolAdapter.handleUpEvent(MessageDispatcher.j
ava:691)
     at
org.jgroups.blocks.MessageDispatcher$ProtocolAdapter.up(MessageDispatcher.java:772)
     at org.jgroups.JChannel.up(JChannel.java:1465)
     at org.jgroups.stack.ProtocolStack.up(ProtocolStack.java:954)
     at org.jgroups.protocols.pbcast.FLUSH.up(FLUSH.java:478)
     at
org.jgroups.protocols.pbcast.STREAMING_STATE_TRANSFER$StateProviderHandler.process(STR
EAMING_STATE_TRANSFER.java:653)
     at
org.jgroups.protocols.pbcast.STREAMING_STATE_TRANSFER$StateProviderThreadSpawner$1.run
(STREAMING_STATE_TRANSFER.java:582)
     at
java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:886)
     at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)
     at java.lang.Thread.run(Thread.java:680)
Caused by: java.util.concurrent.TimeoutException: Could not obtain exclusive
processing lock
     at
org.infinispan.remoting.transport.jgroups.JGroupsDistSync.acquireProcessingLock(JGroup
sDistSync.java:71)
     at
org.infinispan.statetransfer.StateTransferManagerImpl.generateTransactionLog(StateTran
sferManagerImpl.java:202)
     at
org.infinispan.statetransfer.StateTransferManagerImpl.generateState(StateTransferManag
erImpl.java:165)
     ... 12 more
```

This exception is basically saying that the state generator was not able to generate the transaction log and so the output to which it was writing is closed. In this situation, it's common to see the state receiver log an EOFException , as shown below, when trying to read the transaction log because the

sender did not write the transaction log:

```
TRACE [org.infinispan.marshall.VersionAwareMarshaller] (Incoming-2,{brandname}-
Cluster,NodeI-38030:) Log exception reported
java.io.EOFException: Read past end of file
     at
org.jboss.marshalling.AbstractUnmarshaller.eofOnRead(AbstractUnmarshaller.java:184)
     at
org.jboss.marshalling.AbstractUnmarshaller.readUnsignedByteDirect(AbstractUnmarshaller
.java:319)
     at
org.jboss.marshalling.AbstractUnmarshaller.readUnsignedByte(AbstractUnmarshaller.java:
280)
     at
org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshaller.java:207)
     at
org.jboss.marshalling.AbstractUnmarshaller.readObject(AbstractUnmarshaller.java:85)
     at
org.infinispan.marshall.jboss.GenericJBossMarshaller.objectFromObjectStream(GenericJBo
ssMarshaller.java:175)
     at
org.infinispan.marshall.VersionAwareMarshaller.objectFromObjectStream(VersionAwareMars
haller.java:184)
     at
org.infinispan.statetransfer.StateTransferManagerImpl.processCommitLog(StateTransferMa
nagerImpl.java:228)
     at
org.infinispan.statetransfer.StateTransferManagerImpl.applyTransactionLog(StateTransfe
rManagerImpl.java:250)
     at
org.infinispan.statetransfer.StateTransferManagerImpl.applyState(StateTransferManagerI
mpl.java:320)
     at
org.infinispan.remoting.InboundInvocationHandlerImpl.applyState(InboundInvocationHandl
erImpl.java:102)
     at
org.infinispan.remoting.transport.jgroups.JGroupsTransport.setState(JGroupsTransport.j
ava:603)
        ...
```

The current logic is for the state receiver to back off in these scenarios and retry after a few seconds. Quite often, after the retry the state generator might have already finished dealing with the other node and hence the state receiver will be able to fully receive the state.

Why am I getting invalid data passed to readExternal?

If you are using Cache.putAsync() you may find your object is modified after serialization starts, thus corrupting the datastream passed to readExternal . To solve this, make sure you synchronize access to the object.

Read More

You can read more about this issue in this forum thread .

2.9.13. Tuning questions

When running Infinispan under load, I see RejectedExecutionException, how can I fix it?

Internally Infinispan uses executors to do some processing asynchronously, so the first thing to do is to figure out which of these executors is causing issues. For example, if you see a stacktrace that looks like this, the problem is located in the asyncTransportExecutor :

```
java.util.concurrent.RejectedExecutionException
  at
java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution(ThreadPoolExecut
or.java:1759)
  at java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:767)
  at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:658)
  at
java.util.concurrent.AbstractExecutorService.submit(AbstractExecutorService.java:92)
  at
org.infinispan.remoting.transport.jgroups.CommandAwareRpcDispatcher.invokeRemoteComman
ds(CommandAwareRpcDispatcher.java:117)
...
```

To solve this issue, you should try any of these options:

- Increase the maxThreads property in asyncTransportExecutor . At the time of writing, the default value for this particular executor is 25.
- Define your own ExecutorFactory which creates an executor with a bigger queue. You can find more information about different queueing strategies in ThreadPoolExecutor javadoc .
- Disable async marshalling (see the <async ... > element for details). This would mean that an executor is *not* used when replicating, so you will never have a RejectedExecutionException . However this means each put() will take a little longer since marshalling will now happen on the critical path. The RPC is still async though as the thread won't wait for a response from the recipient (fire-and-forget).

2.9.14. JNDI questions

Can I bind Cache or CacheManager to JNDI?

Cache or CacheManager can be bound to JNDI, but only to the java: namespace because they are not designed to be exported outside the Java Virtual Machine. In other words, you shouldn't expect that you'll be able to access them remotely by binding them to JNDI and downloading a remote proxy to them because neither Cache nor CacheManager are serializable.

To find an example on how to bind Cache or CacheManager to the java: namespace, simply check this unit test case .

2.9.15. Hibernate 2nd Level Cache questions

Can I use Infinispan as a remote JPA or Hibernate second level cache?

See Remote Infinispan Caching section in Hibernate documentation.

What are the pitfalls of not using a non-JTA transaction factory such as JDBCTransactionFactory with Hibernate when Infinispan is used as 2nd level cache provider?

The problem is that Hibernate will create a Transaction instance via java.sql.Connection and Infinispan will create a transaction via whatever TransactionManager returned by hibernate.transaction.manager_lookup_class . If hibernate.transaction.manager_lookup_class has not been populated, it will default to the dummy transaction manager.

So, any work on the 2nd level cache will be done under a different transaction to the one used to commit the stuff to the database via Hibernate. In other words, your operations on the database and the 2LC are not treated as a single unit. Risks here include failures to update the 2LC leaving it with stale data while the database committed data correctly.

2.9.16. Cache Server questions

Is there a way to do a Bulk Get on a remote cache?

There's no bulk get operation in Hot Rod, but the Java Hot Rod client has implemented via RemoteCache the getAsync() operation, which returns a org.infinispan.util.concurrent.NotifyingFuture (extends java.util.concurrent.Future). So, if you want to retrieve multiple keys in parallel, just call multiple times *getAsync()* and when you need the values, just call *Future.get()* , or attach a *FutureListener* to the *NotifyingFuture* to get notified when the value is ready.

2.9.17. Debugging questions

How can I get Infinispan to show the full byte array? The log only shows partial contents of byte arrays…

Since version 4.1, whenever Infinispan needs to print byte arrays to logs, these are partially printed in order to avoid unnecessarily printing potentially big byte arrays. This happens in situations where either, Infinispan caches have been configured with lazy deserialization, or your running an Memcached or Hot Rod server. So in these cases, only the first 10 positions of the byte array are shown in the logs. If you want Infinispan to show the full byte array in the logs, simply pass the `-Dinfinispan.arrays.debug=true` system property at startup. In the future, this might be controllable at runtime via a JMX call or similar.

Here's an example of log message with a partially displayed byte array:

```
TRACE [ReadCommittedEntry] (HotRodWorker-1-1) Updating entry
(key=CacheKey{data=ByteArray{size=19, hashCode=1b3278a,
array=[107, 45, 116, 101, 115, 116, 82, 101, 112, 108, ..]}}
removed=false valid=true changed=true created=true
value=CacheValue{data=ByteArray{size=19,
array=[118, 45, 116, 101, 115, 116, 82, 101, 112, 108, ..]},
version=281483566645249}]
```

And here's a log message where the full byte array is shown:

```
TRACE [ReadCommittedEntry] (Incoming-2,{brandname}-Cluster,eq-6834) Updating entry
(key=CacheKey{data=ByteArray{size=19, hashCode=6cc2a4,
array=[107, 45, 116, 101, 115, 116, 82, 101, 112, 108, 105, 99, 97, 116, 101, 100, 80,
117, 116]}}
removed=false valid=true changed=true created=true
value=CacheValue{data=ByteArray{size=19,
array=[118, 45, 116, 101, 115, 116, 82, 101, 112, 108, 105, 99, 97, 116, 101, 100, 80,
117, 116]},
version=281483566645249}]
```

2.9.18. Clustering Transport questions

How do I retrieve the clustering physical address?

You can retrieve the physical address via `AdvancedCache.getRpcManager().getTransport().getPhysicalAddresses()`

2.9.19. Security questions

Using Kerberos with the IBM JDK

When using Kerberos/GSSAPI authentication over Hot Rod, the IBM JDK implementation sometimes fail to authenticate with the following exception:

```
com.ibm.security.krb5.KrbException, status code: 101
    message: Invalid option in ticket request
    at com.ibm.security.krb5.KrbTgsReq.<init>(KrbTgsReq.java:62)
    at com.ibm.security.krb5.KrbTgsReq.<init>(KrbTgsReq.java:145)
    at com.ibm.security.krb5.internal.k.b(k.java:179)
    at com.ibm.security.krb5.internal.k.a(k.java:215)
```

A possible workaround is to perform a login/logout/login on the LoginContext, before using the Subject:

```
LoginContext lc = ...;
lc.login();
lc.logout();
lc = ...;
lc.login();
lc.getSubject();
```

2.10. 2-phase commit

2-phase commit protocol (2PC) is a consensus protocol used for atomically commit or rollback distributed transactions.

More resources

- Wikipedia article

2.11. Atomicity, Consistency, Isolation, Durability (ACID)

According to Wikipedia, ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction.

More resources

- Wikipedia

2.12. Basically Available, Soft-state, Eventually-consistent (BASE)

BASE, also known as Eventual Consistency, is seen as the polar opposite of *ACID*, properties seen as desirable in traditional database systems.

BASE essentially embraces the fact that true consistency cannot be achieved in the real world, and as such cannot be modelled in highly scalable distributed systems. BASE has roots in Eric Brewer's *CAP Theorem*, and eventual consistency is the underpinning of any distributed system that aims to provide high availability and partition tolerance.

Infinispan has traditionally followed ACID principles as far as possible, however an eventually consistent mode embracing BASE is on the roadmap.

More resources

- A good article on ACM compares BASE versus ACID.
- An excellent talk on eventual consistency and BASE in Riak is also available on InfoQ.

2.13. Consistency, Availability and Partition-tolerance (CAP) Theorem

Made famous by Eric Brewer at UC Berkeley, this is a theorem of distributed computing that can be simplified to state that one can only practically build a distributed system exhibiting any two of the three desirable characteristics of distributed systems, which are: Consistency, Availability and Partition-tolerance (abbreviated to CAP). The theorem effectively stresses on the unreliability of networks and the effect this unreliability has on predictable behavior and high availability of dependent systems.

Infinispan has traditionally been biased towards Consistency and Availability, sacrificing Partition-tolerance. However, Infinispan does have a Partition-tolerant, eventually-consistent mode in the pipeline. This optional mode of operation will allow users to tune the degree of consistency they expect from their data, sacrificing partition-tolerance for this added consistency.

More resources

- The theorem is well-discussed online, with many good resources to follow up on, including this document.
- A more recent article by Eric Brewer himself appears on InfoQ a modern analysis of the theorem .

2.14. Consistent Hash

A technique of mapping keys to servers such that, given a stable cluster topology, any server in the cluster can locate where a given key is mapped to with minimal computational complexity.

Consistent hashing is a purely algorithmic technique, and doesn't rely on any metadata or any network broadcasts to "search" for a key in a cluster. This makes it extremely efficient to use.

More resources

- Wikipedia

2.15. Data grid

A data grid is a cluster of (typically commodity) servers, normally residing on a single local-area network, connected to each other using IP based networking. Data grids behave as a single resource, exposing the aggregate storage capacity of all servers in the cluster. Data stored in the grid is usually partitioned, using a variety of techniques, to balance load across all servers in the cluster as evenly as possible. Data is often redundantly stored in the grid to provide resilience to individual servers in the grid failing i.e. more than one copy is stored in the grid, transparently to the application.

Data grids typically behave in a peer-to-peer fashion. Infinispan, for example, makes use of JGroups as a group communication library and is hence biased towards a peer-to-peer design. Such design allows Infinispan to exhibit self-healing characteristics, providing service even when individual servers fail and new nodes are dynamically added to the grid.

Infinispan also makes use of TCP and optionally UDP network protocols, and can be configured to make use of IP multicast for efficiency if supported by the network.

2.16. Deadlock

A deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does.

2.17. Distributed Hash Table (DHT)

A distributed hash table (DHT) is a class of a decentralized distributed system that provides a lookup service similar to a hash table; (key, value) pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

2.18. Externalizer

An *Externalizer* is a class that knows how to marshall a given object type to a byte array, and how to unmarshall the contents of a byte array into an instance of the object type. Externalizers are effectively an Infinispan extension that allows users to specify how their types are serialized. The underlying Infinispan marshalling infrastructure builds on JBoss Marshalling , and offers efficient payloads and stream caching. This provides much better performance than standard Java serialization.

2.19. Hot Rod

Hot Rod is the name of Infinispan's custom TCP client/server protocol which was created in order to overcome the deficiencies of other client/server protocols such as Memcached. HotRod, as opposed to other protocols, has the ability of handling failover on an Infinispan server cluster that undergoes a topology change. To achieve this, the Hot Rod regularly informs the clients of the cluster topology.

Hot Rod enables clients to do smart routing of requests in partitioned, or distributed, Infinispan server clusters. This means that Hot Rod clients can determine the partition in which a key is located and communicate directly with the server that contains the key. This is made possible by Infinispan servers sending the cluster topology to clients, and the clients using the same consistent hash as the servers.

2.20. In-memory data grid

An in-memory data grid (IMDG) is a special type of data grid. In an IMDG, each server uses its main system memory (RAM) as primary storage for data (as opposed to disk-based storage). This allows for much greater concurrency, as lock-free STM techniques such as compare-and-swap can be used to allow hardware threads accessing concurrent datasets. As such, IMDGs are often considered far

better optimized for a multi-core and multi-CPU world when compared to disk-based solutions. In addition to greater concurrency, IMDGs offer far lower latency access to data (even when compared to disk-based data grids using solid state drives).

The tradeoff is capacity. Disk-based grids, due to the far greater capacity of hard disks, exhibit two (or even three) orders of magnitude greater capacity for the same hardware cost.

2.21. Isolation level

Isolation is a property that defines how/when the changes made by one operation become visible to other concurrent operations. Isolation is one of the *ACID* properties.

Infinispan ships with REPEATABLE_READ and READ_COMMITTED isolation levels, the latter being the default.

2.22. JTA synchronization

A Synchronization is a listener which receives events relating to the transaction lifecycle. A Synchronization implementor receives two events, *before completion* and *after completion* . Synchronizations are useful when certain activities are required in the case of a transaction completion; a common usage for a Synchronization is to flush an application's caches.

2.23. Livelock

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing.

A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

2.24. Memcached

Memcached is an in-memory caching system, often used to speed-up database-driven websites. Memcached also defines a text based, client/server, caching protocol, known as the Memcached protocol Infinispan offers a server which speaks the Memcached protocol, allowing Memcached itself to be replaced by Infinispan. Thanks to Infinispan's clustering capabilities, it can offer data failover capabilities not present in original Memcached systems.

2.25. Multiversion Concurrency Control (MVCC)

Multiversion concurrency control is a concurrency control method commonly used by database management systems to provide concurrent access to the database and in programming languages to implement transactional memory.

More resources

- Wikipedia

2.26. Near Cache

A technique for caching data in the client when communicating with a remote cache, for example, over the *Hot Rod* protocol. This technique helps minimize remote calls to retrieve data.

2.27. Network partition

Network partitions happens when multiple parts of a cluster become separated due to some type of network failure, whether permanent or temporary. Often temporary failures heal spontaneously, within a few seconds or at most minutes, but the damage that can occur during a network partition can lead to inconsistent data. Closely tied to Brewer's CAP theorem, distributed systems choose to deal with a network partition by either sacrificing availability (either by shutting down or going into read-only mode) or consistency by allowing concurrent and divergent updates to the same data.

Network partitions are also commonly known as a *Split Brain*, after the biological condition of the same name.

For more detailed discussion, see this blog post.

2.28. NoSQL

A NoSQL database provides a mechanism for storage and retrieval of data that employs less constrained consistency models than traditional relational databases. Motivations for this approach include simplicity of design, horizontal scaling and finer control over availability. NoSQL databases are often highly optimized key–value stores intended for simple retrieval and appending operations, with the goal being significant performance benefits in terms of latency and throughput. NoSQL databases are finding significant and growing industry use in big data and real-time web applications.

2.29. Optimistic locking

Optimistic locking is a concurrency control method that assumes that multiple transactions can complete without affecting each other, and that therefore transactions can proceed without locking the data resources that they affect. Before committing, each transaction verifies that no other transaction has modified its data. If the check reveals conflicting modifications, the committing transaction rolls back.

2.30. Pessimistic locking

A lock is used when multiple threads need to access data concurrently. This prevents data from being corrupted or invalidated when multiple threads try to modify the same item of data. Any single thread can only modify data to which it has applied a lock that gives them exclusive access to the record until the lock is released. However, pessimistic locking isn't ideal from a throughput perspective, as locking is expensive and serializing writes may not be desired. *Optimistic locking* is often seen as a preferred alternative in many cases.

2.31. READ COMMITTED

READ_COMMITTED is one of two isolation levels the Infinispan's locking infrastructure provides (the other is REPEATABLE_READ). Isolation levels have their origins in relational databases.

In Infinispan, READ_COMMITTED works slightly differently to databases. READ_COMMITTED says that "data can be read as long as there is no write", however in Infinispan, reads can happen anytime thanks to *MVCC*. MVCC allows writes to happen on copies of data, rather than on the data itself. Thus, even in the presence of a write, reads can still occur, and all read operations in Infinispan are non-blocking (resulting in increased performance for the end user). On the other hand, write operations are exclusive in Infinispan, (and so work the same way as READ_COMMITTED does in a database).

With READ_COMMITTED, multiple reads of the same key within a transaction can return different results, and this phenomenon is known as non-repeatable reads. This issue is avoided with REPETEABLE_READ isolation level.

2.32. Relational Database Management System (RDBMS)

A relational database management system (RDBMS) is a database management system that is based on the relational model. Many popular databases currently in use are based on the relational database model.

2.33. REPEATABLE READ

REPEATABLE_READ is one of two isolation levels the Infinispan's locking infrastructure provides (the other is READ_COMMITTED). Isolation levels have their origins in relational databases.

In Infinispan, REPEATABLE_READ works slightly differently to databases. REPEATABLE_READ says that "data can be read as long as there are no writes, and vice versa". This avoids the non-repeatable reads phenomenon, because once data has been written, no other transaction can read it, so there's no chance of re-reading the data and finding different data.

Some definitions of REPEATABLE_READ say that this isolation level places shared locks on read data; such lock could not be acquired when the entry is being written. However, Infinispan has an MVCC concurrency model that allows it to have non-blocking reads. Infinispan provides REPEATABLE_READ semantics by keeping the previous value whenever an entry is modified. This allows Infinispan to retrieve the previous value if a second read happens within the same transaction, but it allows following phenomena:

```
cache.get("A") // returns 1
cache.get("B") // returns 1

Thread1: tx1.begin()
Thread1: cache.put("A", 2)
Thread1: cache.put("B", 2)
Thread2:                                    tx2.begin()
Thread2:                                    cache.get("A") // returns 1
Thread1: tx1.commit()
Thread2:                                    cache.get("B") // returns 2
Thread2:                                    tx2.commit()
```

By default, Infinispan uses REPEATABLE_READ as isolation level.

2.34. Representational State Transfer (ReST)

ReST is a software architectural style that promotes accessing resources via a uniform generic interface. HTTP is an implementation of this architecture, and generally when ReST is mentioned, it refers to ReST over HTTP protocol. When HTTP is used, the uniform generic interface for accessing resources is formed of GET, PUT, POST, DELETE and HEAD operations.

Infinispan's ReST server offers a ReSTful API based on these HTTP methods, and allow data to be stored, retrieved and deleted.

2.35. Split brain

A colloquial term for a *network partition*. See *network partition* for more details.

2.36. Structured Query Language (SQL)

SQL is a special-purpose programming language designed for managing data held in a relational database management system (RDBMS). Originally based upon relational algebra and tuple relational calculus, SQL consists of a data definition language and a data manipulation language. The scope of SQL includes data insert, query, update and delete, schema creation and modification, and data access control.

2.37. Write-behind

Write-behind is a cache store update mode. When this mode is used, updates to the cache are asynchronously written to the cache store. Normally this means that updates to the cache store are not performed in the client thread.

An alternative cache store update mode is *write-through*.

2.38. Write skew

In a write skew anomaly, two transactions (T1 and T2) concurrently read an overlapping data set (e.g. values V1 and V2), concurrently make disjoint updates (e.g. T1 updates V1, T2 updates V2), and finally concurrently commit, neither having seen the update performed by the other. Were the system serializable, such an anomaly would be impossible, as either T1 or T2 would have to occur "first", and be visible to the other. In contrast, snapshot isolation such as REPEATABLE_READ and READ_COMMITTED permits write skew anomalies.

Infinispan can detect write skews and can be configured to roll back transactions when write skews are detected.

2.39. Write-through

Write-through is a cache store update mode. When this mode is used, clients update a cache entry, e.g. via a Cache.put() invocation, the call will not return until Infinispan has updated the underlying cache store. Normally this means that updates to the cache store are done in the client thread.

An alternative mode in which cache stores can be updated is *write-behind*.

2.40. XA resource

An XA resource is a participant in an XA transaction (also known as a distributed transaction). For example, given a distributed transaction that operates over a database and Infinispan, XA defines both Infinispan and the database as XA resources.

Java's API for XA transactions is JTA and XAResource is the Java interface that describes an XA resource.

2.41. Upgrading from 9.4 to 10.0

2.41.1. Hot Rod 3.0

Older versions of the Hot Rod protocol treated expiration values greater than the number of milliseconds in 30 days as Unix time. Starting with Hot Rod 3.0 this adjustment no longer happens and expiration is taken literally.

2.41.2. Total Order transaction protocol is deprecated

Total Order transaction protocol is going to be removed in a future release. Use the default protocol (2PC).

2.41.3. Removed the infinispan.server.hotrod.workerThreads system property

The `infinispan.server.hotrod.workerThreads` property was introduced as a hack to work around the fact that the configuration did not expose it. The property has been removed and endpoint worker

threads must now be exclusively configured using the `worker-threads` attribute.

2.41.4. Removed AtomicMap and FineGrainedAtomicMap

AtomicMapLookup, AtomicMap and FineGrainedAtomicMap have been removed. Please see FunctionalMaps or Cache#Merge for similar functionality.

2.41.5. Removed Delta and DeltaAware

The previously deprecated Delta and DeltaAware interfaces have been removed.

2.41.6. Removed compatibility mode

The previously deprecated Compatibility Mode has been removed.

2.41.7. Removed the implicit default cache

The default cache must now be named explicitly via the GlobalConfigurationBuilder#defaultCacheName() method.

2.41.8. Removed DistributedExecutor

The previously deprecated DistributedExecutor is now removed. References should be updated to use ClusterExecutor.

2.41.9. Removed the Tree module

TreeCache has been unsupported for a long time and was only intended as a quick stopgap for JBossCache users. The module has now been removed completely.

2.41.10. The JDBC PooledConnectionFactory now utilises Agroal

Previously the JDBC PooledConnectionFactory provided c3p0 and HikariCP based connection pools. From 10.0 we only provide a PooledConnectionFactory based upon the Agroal project. This means that it is no longer possible to utilise `c3p0.properties` and `hikari.properties` files to configure the pool, instead an agroal compatiblet properties file can be provided.

2.41.11. XML configuration changes

Several configuration elements and attributes that were deprecated since 9.0 have been removed:

- `<eviction>` - replaced with `memory`
- `<versioning>` - automatically enabled
- `<data-container>` - no longer customizable
- `deadlock-detection-spin` - always disabled
- `write-skew` - enabled automatically

2.41.12. RemoteCache Changes

The getBulk methods have been removed

The getBulk method is an expensive method as it requires holding all keys in memory at once and requires a possibly very single result to populate it. The new retrieveEntries, entrySet, keySet and values methods handle this in a much more efficient way. Therefore the getBulk methods have been removed in favor of them.

2.41.13. Persistence changes

- File-based cache stores (SingleFileStore, SoftIndexFileStore, RocksDBStore) filesystem layout has been normalized so that they will use the `GlobalStateConfiguration` persistent location as a default location. Additionally, all stores will now use the cache name as part of the data file/directory naming allowing multiple stores to avoid conflicts and ambiguity.
- The CLI loader (`infinispan-persistence-cli`) has been removed.
- The LevelDB store (`infinispan-cachestore-leveldb`) has been removed. Use the RocksDB store instead, as it is fully backwards compatible.
- The deprecated `singleton` store configuration option and the wrapper class `SingletonCacheWriter` have been removed.

  Using `shared=true` is enough, as only the primary owner of each key will write to a shared store.

2.41.14. Client/Server changes

- The Hot Rod client and server only support protocol versions 2.0 and higher. Support for Hot Rod versions 1.0 to 1.3 has been dropped.

2.41.15. SKIP_LISTENER_NOTIFICATION flag

`SKIP_LISTENER_NOTIFICATION` notification flag has been added in the hotrod client. This flag only works when the client and the server version is 9.4.15 or higher. Spring Session integration uses this flag when a session id has changed. If you are using Spring Session with Infinispan 9.4, consider upgrading the client and the server.

2.41.16. performAsync header removed from REST

The `performAsync` header was removed from the REST server. Clients that want to perform async operations with the REST server should manage the request and response on their side to avoid blocking.

2.41.17. Default JGroups stacks in the XML configuration

With the introduction of inline XML JGroups stacks in the configuration, two default stacks are always enabled: `udp` and `tcp`. If you are declaring your own stacks with the same names, an exception reporting the conflict will be thrown. Simply rename your own configurations to avoid the conflict.

2.41.18. JGroups S3_PING replaced with NATIVE_S3_PING

Because of changes in AWS's access policy regarding signatures, S3_PING will not work in newer regions and will stop working in older regions too. For this reason, you should migrate to using NATIVE_S3_PING instead.

2.41.19. Cache and Cache Manager Listeners can now be configured to be non blocking

Listeners in the past that were sync, always ran in the thread that caused the event. We now allow a Listener method to be non-blocking in that it will still fire in the original thread, under the assumption that it will return immediately. Please read the Listener Javadoc for information and examples on this.

2.41.20. Distributed Streams operations no longer support null values

Distributed Streams has parts rewritten to utilize non blocking reactive streams based operations. As such null values are not supported as values from operations as per the reactive streams spec. Please utilize other means to denote a null value.

2.41.21. Removed the infinispan-cloud module

The infinispan-cloud module has been removed and the `kubernetes`, `ec2`, `google` and `azure` default configurations have been included in `infinispan-core` and can be referenced as default named JGroups configurations.

2.41.22. Removed experimental flag GUARANTEED_DELIVERY

Almost as soon as GUARANTEED_DELIVERY was added, UNICAST3 and NAKACK2.resend_last_seqno removed the need for it. It was always documented as experimental, so we removed it without deprecation and we also removed the RSVP protocol from the default JGroups stacks.

2.41.23. Cache Health

The possible statuses of the cache health are now HEALTHY, HEALTHY_REBALANCING and DEGRADED to better reflect the fact that `rebalancing` doesn't mean a cluster is unhealthy.

2.41.24. Multi-tenancy

When using multi-tenancy in the Wildfly based server, it's necessary to specify the `content-path` for each of the REST connectors, to match the `prefix` element under `multi-tenancy\rest\prefix`.

2.41.25. OffHeap Automatic Resizing

Off Heap memory containers now will dynamically resize based on number of entries in the container. Due to this the address count configuration value is now deprecated for APIs and has been removed from the xml parser.

2.42. Upgrading from 9.3 to 9.4

2.42.1. Client/Server changes

Compatibility mode deprecation

Compatibility mode has been deprecated and will be removed in the next Infinispan version.

To use a cache from multiple endpoints, it is recommended to store data in binary format and to configure the MediaType for keys and values.

If storing data as unmarshalled objects is still desired, the equivalent of compatibility mode is to configure keys and values to store object content:

```
<encoding>
   <key media-type="application/x-java-object"/>
   <value media-type="application/x-java-object"/>
</encoding>
```

Memcached storage

For better interoperability between endpoints, the Memcached server no longer stores keys as `java.lang.String`, but as UTF-8 `byte[]`.

If using memcached, it's recommended to run a rolling upgrade from 9.3 to store data in the new format, or reload the data in the cache.

Scripts Response

Distributed scripts with text-based data type no longer return `null` when the result from each server is null. The response is now a JSON array with each individual result, e.g. `"[null, null]"`

WebSocket endpoint removal

The WebSocket endpoint has been unmaintained for several years. It has been removed.

Hot Rod client connection pool properties

Since the Hot Rod client was overhauled in 9.2, the way the connection pool configuration is handled has changed. Infinispan 9.4 introduces a new naming scheme for the connection pool properties which deprecates the old *commons-pool* names. For a complete reference of the available configuration options for the properties file please refer to remote client configuration javadoc.

Server thread pools

The threads that handle the child Netty event loops have been renamed from *-ServerWorker to *-ServerIO

2.42.2. Persistence Changes

Shared and Passivation

A store cannot be configured as both shared and having passivation enabled. Doing so can cause data inconsistencies as there is no way to synchronize data between all the various nodes. As such this configuration will now cause a startup exception. Please update your configuration as appropriate.

2.42.3. Query changes

AffinityIndexManager

The default number of shards is down to 4, it was previously equals to the number of segments in the cache.

2.43. Upgrading from 9.2 to 9.3

2.43.1. AdvancedCacheLoader changes

The AdvancedCacheLoader SPI has been enhanced to provide an alternative method to process and instead allows reactive streams based publishKeys and publishEntries methods which provide benefits in performance, threading and ease of use. Note this change will only affect you if you wish take advantage of it in any custom CacheLoaders you may have implemented.

2.43.2. Partition Handling Configuration

In 9.3 the default MergePolicy is now MergePolicy.NONE, opposed to MergePolicy.PREFERRED_ALWAYS.

2.43.3. Stat Changes

We have reverted the stat changes introduced in 9.1, so average values for read, write and removals are once again returned as milliseconds.

2.43.4. Event log changes

Several new event log messages have been added, and one message has been removed (ISPN100013).

2.43.5. Max Idle Expiration Changes

The max idle entry expiration information is sent between owners in the cluster. However when an entry expires via max idle on a given node, this was not replicated (only removing it locally). Max idle has been enhanced to now expire an entry across the entire cluster, instead of per node. This includes ensuring that max idle expiration is applied across all owners (meaning if another node has accessed the entry within the given time it will prevent that entry from expiring on other nodes that didn't have an access).

Max idle in a transactional clustered cache does not remove expired entries on access (although it will not be returned). These entries are only removed via the expiration reaper.

Iteration in a clustered cache will still show entries that are expired via maxIdle to ensure good performance, but could be removed at any point due to expiration reaper.

2.43.6. Wildfly Modules

The Infinispan Wildfly modules are now located in the `system/add-ons/{moduleprefix}` dir as per the Wildfly module conventions.

2.43.7. Deserialization Whitelist

Deserialization of content sent by clients to the server are no longer allowed by default. This applies to JSON, XML, and marshalled byte[] that, depending on the cache configuration, will cause the server to convert it to Java Objects either to store it or to perform any operation that cannot be done on a byte[] directly.

The deserialization needs to be enabled using system properties, ether by class name or regular expressions:

```
// Comma separated list of fully qualified class names
-Dinfinispan.deserialization.whitelist.classes=java.time.Instant,com.myclass.Entity

// Regex expression
-Dinfinispan.deserialization.whitelist.regexps=.*
```

2.44. Upgrading from 9.0 to 9.1

2.44.1. Kubernetes Ping changes

The latest version of Kubernetes Ping uses unified environmental variables for both Kubernetes and OpenShift. Some of them were shortened for example `OPENSHIFT_KUBE_PING_NAMESPACE` was changed to `KUBERNETES_NAMESPACE`. Please refer to Kubernetes Ping documentation.

2.44.2. Stat Changes

Average values for read, write and removals are now returned in Nanoseconds, opposed to Milliseconds.

2.44.3. (FineGrained)AtomicMap reimplemented

Infinispan now contains a new implementation of both `AtomicMap` and `FineGrainedAtomicMap`, but the semantics has been preserved. The new implementation does not use `DeltaAware` interface but the Functional API instead.

There are no changes needed for `AtomicMap`, but it now supports non-transactional use case as well.

`FineGrainedAtomicMap` now uses the Grouping API and therefore you need to enable groups in configuration. Also it holds entries as regular cache entries, plus one cache entry for cached key set (the map itself). Therefore the cache size or iteration/streaming results may differ. Note that fine grained atomic maps are still supported on transactional caches only.

2.44.4. RemoteCache keySet/entrySet/values

RemoteCache now implements all of the collection backed methods from `Map` interface. Previously `keySet` was implemented, however it was a deep copy. This has now changed and it is a backing set. That is that the set retrieves the updated values on each invocation or updates to the backing remote cache for writes. The entrySet and values methods are also now supported as backing variants as well.

If you wish to have a copy like was provided before it is recommended to copy the contents into a in memory local set such as

```
Set<K> keysCopy = remoteCache.keySet().stream().collect(Collectors.toSet());
```

2.44.5. DeltaAware deprecated

Interfaces `DeltaAware`, `Delta` and `CopyableDeltaAware` have been deprecated. Method `AdvancedCache.applyDelta()` has been deprecated and the implementation does not allow custom set of locked keys. `ApplyDeltaCommand` and its uses in interceptor stack are deprecated.

Any partial updates to an entry should be replaced using the Functional API.

2.44.6. Infinispan Query Configuration

The configuration property `directory_provider` now accepts a new value `local-heap`. This value replaces the now deprecated `ram`, and as its predecessor will cause the index to be stored in a `org.apache.lucene.store.RAMDirectory`.

The configuration value `ram` is still accepted and will have the same effect, but failing to replace `ram` with `local-heap` will cause a warning to be logged. We suggest to perform this replacement, as the `ram` value will no longer be recognised by Infinispan in a future version.

This change was made as the team believes the `local-heap` name better expresses the storage model, especially as this storage method will not allow real-time replication of the index across multiple nodes. This index storage option is mostly useful for single node integration testing of the query functionality.

2.44.7. Store Batch Size Changes

TableManipulation::batchSize and JpaStoreConfiguration::batchSize have been deprecated and replaced by the higher level AbstractStoreConfiguration::maxBatchSize.

2.44.8. Partition Handling changes

In Infinispan 9.1 partition handling has been improved to allow for automatic conflict resolution on partition merges. Consequently, PartitionHandlingConfiguration::enabled has been deprecated in favour of PartitionHandlingConfiguration::whenSplit. Configuring whenSplit to the DENY_READ_WRITES strategy is equivalent to setting enabled to true, whilst specifying ALLOW_READ_WRITES is equivalent to disabling partition handling (default).

Furthermore, during a partition merge with ALLOW_READ_WRITES, the default EntryMergePolicy is MergePolicies.PREFERRED_ALWAYS which provides a deterministic way of tie-breaking CacheEntry conflicts. If you require the old behaviour, simply set the merge-policy to null.

2.45. Upgrading from 8.x to 9.0

2.45.1. Default transaction mode changed

The default configuration for transactional caches changed from `READ_COMMITTED` and `OPTIMISTIC` locking to `REPEATABLE_READ` and `OPTIMISTIC` locking with `write-skew` enabled.

Also, using the `REPEATABLE_READ` isolation level and `OPTIMISTIC` locking without `write-skew` enabled is no longer allowed. To help with the upgrade, `write-skew` will be automatically enabled in this case.

The following configuration has been deprecated:

- `write-skew`: as said, it is automatically enabled.
- `<versioning>` and its attributes. It is automatically enabled and configured when needed.

2.45.2. Removed eagerLocking and eagerLockingSingleNode configuration settings

Both were deprecated since version 5.1. `eagerLocking(true)` can be replaced with `lockingMode(LockingMode.PESSIMISTIC)`, and `eagerLockingSingleNode()` does not need a replacement because it was a no-op.

2.45.3. Removed async transaction support

Asynchronous mode is no longer supported in transactional caches and it will automatically use the synchronous cache mode. In addition, the second phase of a transaction commit is done synchronously. The following methods (and related) are deprecated:

- `TransactionConfigurationBuilder.syncCommitPhase(boolean)`
- `TransactionConfigurationBuilder.syncRollbackPhase(boolean)`

2.45.4. Deprecated all the dummy related transaction classes.

The following classes have been deprecated and they will be removed in the future:

- `DummyBaseTransactionManager`: replaced by `EmbeddedBasedTransactionManager`;

- `DummyNoXaXid` and `DummyXid`: replaced by `EmbeddedXid`;
- `DummyTransaction`: replaced by `EmbeddedTransaction`;
- `DummyTransactionManager`: replaced by `EmbeddedTransactionManager`;
- `DummyTransactionManagerLookup` and `RecoveryDummyTransactionManagerLookup`: replaced by `EmbeddedTransactionManagerLookup`;
- `DummyUserTransaction`: replaced by `EmbeddedUserTransaction`;

2.45.5. Clustering configuration changes

The `mode` attribute in the XML declaration of clustered caches is no longer mandatory. It defaults to SYNC.

2.45.6. Default Cache changes

Up to Infinispan 8.x, the default cache always implicitly existed, even if not declared in the XML configuration. Additionally, the default cache configuration affected all other cache configurations, acting as some kind of base template. Since 9.0, the default cache only exists if it has been explicitly configured. Additionally, even if it has been specified, it will never act as base template for other caches.

2.45.7. Marshalling Enhancements and Store Compatibility

Internally Infinispan 9.x has introduced many improvements to its marshalling codebase in order to improve performance and allow for greater flexibility. Consequently, data marshalled and persisted by Infinispan 8.x is no longer compatible with Infinispan 9.0. To aid you in migrating your existing stores to 9.0, we have provided a Store Migrator, however at present this only allows the migration of JDBC stores.

2.45.8. New Cloud module for library mode

In Infinispan 8.x, cloud related configuration were added to `infinispan-core` module. Since 9.0 they were moved to `infinispan-cloud` module.

2.45.9. Entry Retriever is now removed

The entry retriever feature has been removed. Please update to use the new Streams feature detailed in the User Guide. The `org.infinispan.filter.CacheFilters` class can be used to convert `KeyValueFilter` and `Converter` instances into proper Stream operations that are able to be marshalled.

2.45.10. Map / Reduce is now removed

Map reduce has been removed in favor of the new Streams feature which should provide more features and performance. There are no bridge classes to convert to the new streams and all references must be rewritten.

2.45.11. Spring 4 support is now removed

Spring 4 is no longer supported.

2.45.12. Function classes have moved packages

The class `SerializableSupplier` has moved from the `org.infinispan.stream` package to the `org.infinispan.util.function` package.

The class `CloseableSupplier` has moved from the `org.infinispan.util` package to the `org.infinispan.util.function` package.

The classes `TriConsumer`, `CloseableSupplier`, `SerializableRunnable`, `SerializableFunction` & `SerializableCallable` have all been moved from the `org.infinispan.util` package to the `org.infinispan.util.function` package.

2.45.13. SegmentCompletionListener interface has moved

The interface `SegmentCompletionListener` has moved from the interface `org.infinispan.CacheStream` to the new `org.infinispan.BaseCacheStream`.

2.45.14. Spring module dependency changes

All Infinispan, Spring and Logger dependencies are now in the `provided` scope. One can decide whether to use small jars or uber jars but they need to be added to the classpath of the application. It also gives one freedom in choosing Spring (or Spring Boot) version.

Here is an example:

```
<dependencies>
   <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-embedded</artifactId>
   </dependency>
   <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-spring5-embedded</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework.session</groupId>
      <artifactId>spring-session</artifactId>
   </dependency>
</dependencies>
```

Additionally there is no Logger implementation specified (since this may vary depending on use

case).

2.45.15. Total order executor is now removed

The total order protocol now uses the `remote-command-executor`. The attribute `total-order-executor` in `<container>` tag is removed.

2.45.16. HikariCP is now the default implementation for JDBC PooledConnectionFactory

HikariCP offers superior performance to c3p0 and is now the default implementation. Additional properties for HikariCP can be provided by placing a `hikari.properties` file on the classpath or by specifying the path to the file via `PooledConnectionFactoryConfiguration.propertyFile` or `properties-file` in the connection pool's xml config. N.B. a properties file specified explicitly in the configuration is loaded instead of the `hikari.properties` file on the class path and Connection pool characteristics which are explicitly set in PooledConnectionFactoryConfiguration always override the values loaded from a properties file.

Support for c3p0 has been deprecated and will be removed in a future release. Users can force c3p0 to be utilised as before by providing the system property `-Dinfinispan.jdbc.c3p0.force=true`.

2.45.17. RocksDB in place of LevelDB

The LevelDB cache store was replaced with a RocksDB. RocksDB is a fork of LevelDB which provides superior performance in high concurrency scenarios. The new cache store can parse old LevelDB configurations but will always use the RocksDB implementation.

2.45.18. JDBC Mixed and Binary stores removed

The JDBC Mixed and Binary stores have been removed due to the poor performance associated with storing entries in buckets. Storing entries in buckets is non-optimal as each read/write to the store requires an existing bucket for a given hash to be retrieved, deserialised, updated, serialised and then re-inserted back into the db. If you were previously using one of the removed stores, we have provided a migrator tool to assist in migrating data from an existing binary table to a JDBC string based store.

2.45.19. @Store Annotation Introduced

A new annotation, `@Store`, has been added for persistence stores. This allows a store's properties to be explicitly defined and validated against the provided store configuration. Existing stores should be updated to use this annotation and the store's configuration class should also declare the `@ConfigurationFor` annotation. If neither of these annotations are present on the store or configuration class, then a your store will continue to function as before, albeit with a warning that additional store validation cannot be completed.

2.45.20. Server authentication changes

The no-anonymous policy is now automatically enabled for Hot Rod authentication unless explicitly specified.

2.45.21. Package org.infinispan.util.concurrent.jdk8backported has been removed

Moved classes

Classes regarding EntrySizeCalculator have now been moved down to the org.infinispan.util package.

Removed classes

The *ConcurrentHashMapV8 classes and their supporting classes have all been removed. The CollectionFactory#makeBoundedConcurrentMap method should be used if you desire to have a bounded ConcurrentMap.

2.45.22. Store as Binary is deprecated

Store as Binary configuration is now deprecated and will be removed in a future release. This is replaced by the new memory configuration.

2.45.23. DataContainer collection methods are deprecated

The keySet, entrySet and values methods on DataContainer have been deprecated. These behavior of these methods are very inconsistent and will be removed later. It is recommended to update references to use iterator or iteratorIncludingExpired methods intead.

2.46. Upgrading from 8.1 to 8.2

2.46.1. Entry Retriever is deprecated

Entry Retriever is now deprecated and will be removed in Infinispan 9. This is replaced by the new Streams feature.

2.46.2. Map / Reduce is deprecated

Map reduce is now deprecated and will be removed in Infinispan 9. This is replaced by the new Streams feature.

2.47. Upgrading from 8.x to 8.1

2.47.1. Packaging changes

CDI module split

CDI module (GroupId:ArtifactId `org.infinispan:infinispan-cdi`) has been split into `org.infinispan:infinispan-cdi-embedded` and `org.infinispan:infinispan-cdi-remote`. Please make sure that you use proper artifact.

Spring module split

Spring module (GroupId:ArtifactId `org.infinispan:infinispan-spring5`) has been split into `org.infinispan:infinispan-spring5-embedded` and `org.infinispan:infinispan-spring5-remote`. Please make sure that you use proper artifact.

2.47.2. Spring 3 support is deprecated

Spring 3 support (GroupId:ArtifactId `org.infinispan:infinispan-spring`) is deprecated. Please consider migrating into Spring 4 support.

2.48. Upgrading from 7.x to 8.0

2.48.1. Configuration changes

Removal of Async Marshalling

Async marshalling has been entirely dropped since it was never reliable enough. The "async-marshalling" attribute has been removed from the 8.0 XML schema and will be ignored when parsing 7.x configuration files. The programmatic configuration methods related to asyncMarshalling/syncMarshalling are now deprecated and have no effect aside from producing a WARN message in the logs.

Reenabling of isolation level configurations in server

Because of the inability to configure write skew in the server, the isolation level attribute was ignored and defaulted to READ_COMMITTED. Now, when enabling REPEATABLE_READ together with optimistic locking, write skew is enabled by default in local and synchronous configurations.

Subsystem renaming in server

In order to avoid conflict and confusion with the similar subsystems in WildFly, we have renamed the following subsystems in server: * infinispan → datagrid-infinispan * jgroups → datagrid-jgroups * endpoint → datagrid-infinispan-endpoint

Server domain mode

We no longer support the use of standalone mode for running clusters of servers. Domain mode (bin/domain.sh) should be used instead.

2.49. Upgrading from 6.0 to 7.0

2.49.1. API Changes

Cache Loader

To be more inline with JCache and java.util.collections interfaces we have changed the first argument type for the CacheLoader.load & CacheLoader.contains methods to be Object from type K.

Cache Writer

To be more inline with JCache and java.util.collections interfaces we have changed the first argument type for the CacheWriter.delete method to be Object from type K.

Filters

Over time Infinispan added 2 interfaces with identical names and almost identical methods. The org.infinispan.notifications.KeyFilter and org.infinispan.persistence.spi.AdvancedCacheLoader$KeyFilter interfaces.

Both of these interfaces are used for the sole purpose of filtering an entry by it's given key. Infinispan 7.0 has also introduced the KeyValueFilter which is similar to both but also can filter on the entries value and/or metadata.

As such all of these classes have been moved into a new package org.infinispan.filter and all of their related helper classes.

The new org.infinispan.filter.KeyFilter interface has replaced both of the previous interfaces and all previous references use the new interface.

2.49.2. Declarative configuration

The XML schema for the embedded configuration has changed to more closely follow the server configuration. Use the `config-converter.sh` or `config-converter.bat` scripts to convert an Infinispan 6.0 to the current format.

2.50. Upgrading from 5.3 to 6.0

2.50.1. Declarative configuration

In order to use all of the latest features, make sure you change the namespace declaration at the top of your XML configuration files as follows:

```
<infinispan xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"urn:infinispan:config:6.0 http://www.infinispan.org/schemas/infinispan-config-
6.0.xsd" xmlns="urn:infinispan:config:6.0">
   ...
</infinispan>
```

2.50.2. Deprecated API removal

- Class `org.infinispan.persistence.remote.wrapperEntryWrapper`.
- Method `ObjectOutput startObjectOutput(OutputStream os, boolean isReentrant)` from class `org.infinispan.commons.marshall.StreamingMarshaller`.
- Method `CacheEntry getCacheEntry(Object key, EnumSet<Flag> explicitFlags, ClassLoader explicitClassLoader)` from class `org.infinispan.AdvancedCache`. Please use instead:

`AdvanceCache.withFlags(Flag… flags).with(ClassLoader classLoader).getCacheEntry(K key)`.

- Method `AtomicMap<K, V> getAtomicMap(Cache<MK, ?> cache, MK key, FlagContainer flagContainer)` from class `org.infinispan.atomic.AtomicMapLookup`. Please use instead `AtomicMapLookup.getAtomicMap(cache.getAdvancedCache().withFlags(Flag… flags), MK key)`.
- Package `org.infinispan.config` (and all methods involving the old configuration classes). All methods removed has an overloaded method which receives the new configuration classes as parameters. Please refer to [_configuration] for more information about the new configuration classes.

This only affects the programmatic configuration.

- Class `org.infinispan.context.FlagContainer`.
- Method `boolean isLocal(Object key)` from class `org.infinispan.distribution.DistributionManager`. Please use instead `DistributionManager.getLocality(Object key)`.
- JMX operation `void setStatisticsEnabled(boolean enabled)` from class `org.infinispan.interceptors.TxInterceptor` Please use instead the `statisticsEnabled` attribute.
- Method `boolean delete(boolean synchronous)` from class `org.infinispan.io.GridFile`. Please use instead `GridFile.delete()`.
- JMX attribute `long getLocallyInterruptedTransactions()` from class `org.infinispan.util.concurrent.locks.DeadlockDetectingLockManager`.

2.51. Upgrading from 5.2 to 5.3

2.51.1. Declarative configuration

In order to use all of the latest features, make sure you change the namespace declaration at the top of your XML configuration files as follows:

```
<infinispan xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"urn:infinispan:config:5.2 http://www.infinispan.org/schemas/infinispan-config-
5.2.xsd" xmlns="urn:infinispan:config:5.3">
   ...
</infinispan>
```

2.52. Upgrading from 5.1 to 5.2

2.52.1. Declarative configuration

In order to use all of the latest features, make sure you change the namespace declaration at the top of your XML configuration files as follows:

```
<infinispan xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"urn:infinispan:config:5.2 http://www.infinispan.org/schemas/infinispan-config-
5.2.xsd" xmlns="urn:infinispan:config:5.2">
   ...
</infinispan>
```

2.52.2. Transaction

The default transaction enlistment model has changed (ISPN-1284) from `XAResource` to `Synchronization`. Also now, if the `XAResource` enlistment is used, then recovery is enabled by default.

In practical terms, if you were using the default values, this should not cause any backward compatibility issues but an increase in performance of about 5-7%. However in order to use the old configuration defaults, you need to configure the following:

```
<transaction useSynchronization="false">
   <recovery enabled="false"/>
</transaction>
```

or the programmatic configuration equivalent:

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.transaction().useSynchronization(false).recovery().enabled(false)
```

2.52.3. Cache Loader and Store configuration

Cache Loader and Store configuration has changed greatly in Infinispan 5.2.

2.52.4. Virtual Nodes and Segments

The concept of Virtual Nodes doesn't exist anymore in Infinispan 5.2 and has been replaced by Segments.

2.53. Upgrading from 5.0 to 5.1

2.53.1. API

The cache and cache manager hierarchies have changed slightly in 5.1 with the introduction of `BasicCache` and `BasicCacheContainer` , which are parent classes of existing `Cache` and `CacheContainer` classes respectively. What's important is that Hot Rod clients must now code against `BasicCache` and `BasicCacheContainer` rather than `Cache` and `CacheContainer`. So previous code that was written like this will no longer compile.

WontCompile.java

```
import org.infinispan.Cache;
import org.infinispan.manager.CacheContainer;
import org.infinispan.client.hotrod.RemoteCacheManager;
...
CacheContainer cacheContainer = new RemoteCacheManager();
Cache cache = cacheContainer.getCache();
```

Instead, if Hot Rod clients want to continue using interfaces higher up the hierarchy from the remote cache/container classes, they'll have to write:

Correct.java

```
import org.infinispan.BasicCache;
import org.infinispan.manager.BasicCacheContainer;
import org.infinispan.client.hotrod.RemoteCacheManager;
...
BasicCacheContainer cacheContainer = new RemoteCacheManager();
BasicCache cache = cacheContainer.getCache();
```

However, previous code that interacted against the `RemoteCache` and `RemoteCacheManager` will work as it used to:

AlsoCorrect.java

```
import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.client.hotrod.RemoteCacheManager;
...
RemoteCacheManager cacheContainer = new RemoteCacheManager();
RemoteCache cache = cacheContainer.getCache();
```

2.53.2. Eviction and Expiration

- The eviction XML element no longer defines the `wakeUpInterval` attribute. This is now configured via the `expiration` element:

```
<expiration wakeUpInterval="60000"... />
```

Eviction's `maxEntries` is used as guide for the entire cache, but eviction happens on a per cache segment, so when the segment is full, the segment is evicted. That's why `maxEntries` is a theoretical limit but in practical terms, it'll be a bit less than that. This is done for performance reasons.

2.53.3. Transactions

- A cache marked as `TRANSACTIONAL` cannot be accessed outside of a transaction, and a `NON_TRANSACTIONAL` cache cannot be accessed within a transaction. In 5.0, a transactional cache

would support non-transactional calls as well. This change was done to be in-line with expectations set out in JSR-107 as well as to provide more consistent behavior.

- In 5.0, commit and rollback phases were asynchronous by default. Starting with 5.1, these are now synchronous by default, to provide the guarantees required by a single lock-owner model.

2.53.4. State transfer

One of the big changes we made in 5.1 was to use the same push-based state transfer we introduced in 5.0 both for rehashing in distributed mode and for state retrieval in replicated mode. We even borrow the consistent hash concept in replicated mode to transfer state from all previous cache members at once in order to speed up transfer.

As a consequence we've unified the state transfer configuration as well, there is now a `stateTransfer` element containing a simplified state transfer configuration. The corresponding attributes in the `stateRetrieval` and `hash` elements have been deprecated, as have been some attributes that are no longer used.

2.53.5. Configuration

If you use XML to configure Infinispan, you shouldn't notice any change, except a much faster startup, courtesy of the StAX based parser. However, if you use programmatic configuration, read on for the important differences.

Configuration is now packaged in `org.infinispan.configuration`, and you must use a fluent, builder style:

```
Configuration c1 = new ConfigurationBuilder()
   // Adjust any configuration defaults you want
   .clustering()
      .l1()
         .disable()
      .mode(DIST_SYNC)
      .hash()
         .numOwners(5)
   .build();
```

- The old javabean style configuration is now deprecated and will be removed in a later version.
- Configuration properties which can be safely changed at runtime are mutable, and all others are immutable.
- To copy a configuration, use the `read()` method on the builder, for example:

```
Configuration c2 = new ConfigurationBuilder()
   // Read in C1 to provide defaults
   .read(c1)
   .clustering()
      .l1()
         .enable()
   // This cache is DIST_SYNC, will have 5 owners, with L1 cache enabled
   .build();
```

This completely replaces the old system of defining a set of overrides on bean properties. Note that this means the behaviour of Infinispan configuration is somewhat different when used programmatically. Whilst before, you could define a default configuration, and any overrides would be applied on top of *your* defaults when defined, now you must explicitly read in your defaults to the builder. This allows for much greater flexibility in your code (you can have a as many "default" configurations as you want), and makes your code more explicit and type safe (finding references works).

The schema is unchanged from before. Infinispan 4.0 configurations are currently not being parsed. To upgrade, just change the schema definition from:

```
<infinispan
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="urn:infinispan:config:4.1
http://www.infinispan.org/schemas/infinispan-config-4.1.xsd"
     xmlns="urn:infinispan:config:4.1">
```

to

```
<infinispan
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="urn:infinispan:config:5.1
http://www.infinispan.org/schemas/infinispan-config-5.1.xsd"
     xmlns="urn:infinispan:config:5.1">
```

The schema documentation has changed format, as it is now produced using the standard tool `xsddoc`. This should be a significant improvement, as better navigation is offered. Some elements and attributes are missing docs right now, we are working on adding this. As an added benefit, your IDE should now show documentation when an xsd referenced (as above)

We are in the process of adding in support for this configuration style for modules (such as cache stores). In the meantime, please use the old configuration or XML if you require support for cache store module configuration.

2.53.6. Flags and ClassLoaders

The `Flags` and `ClassLoader` API has changed. In the past, the following would work:

```
cache.withFlags(f1, f2); cache.withClassLoader(cl); cache.put(k, v);
```

In 5.1.0, these `withX()` methods return a new instance and not the cache itself, so thread locals are avoided and the code above will not work. If used in a fluent manner however, things still work:

```
cache.withFlags(f1, f2).withClassLoader(cl).put(k, v);
```

The above pattern has always been the intention of this API anyway.

2.53.7. JGroups Bind Address

Since upgrading to JGroups 3.x, `-Dbind.address` is ignored. This should be replaced with `-Djgroups.bind_addr`.