Infinispan Technical Overview

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.

Chapter 3. Client/Server

Infinispan offers two alternative access methods:

- Embedded mode: The Infinispan libraries co-exist with the user application in the same JVM as shown in the following diagram

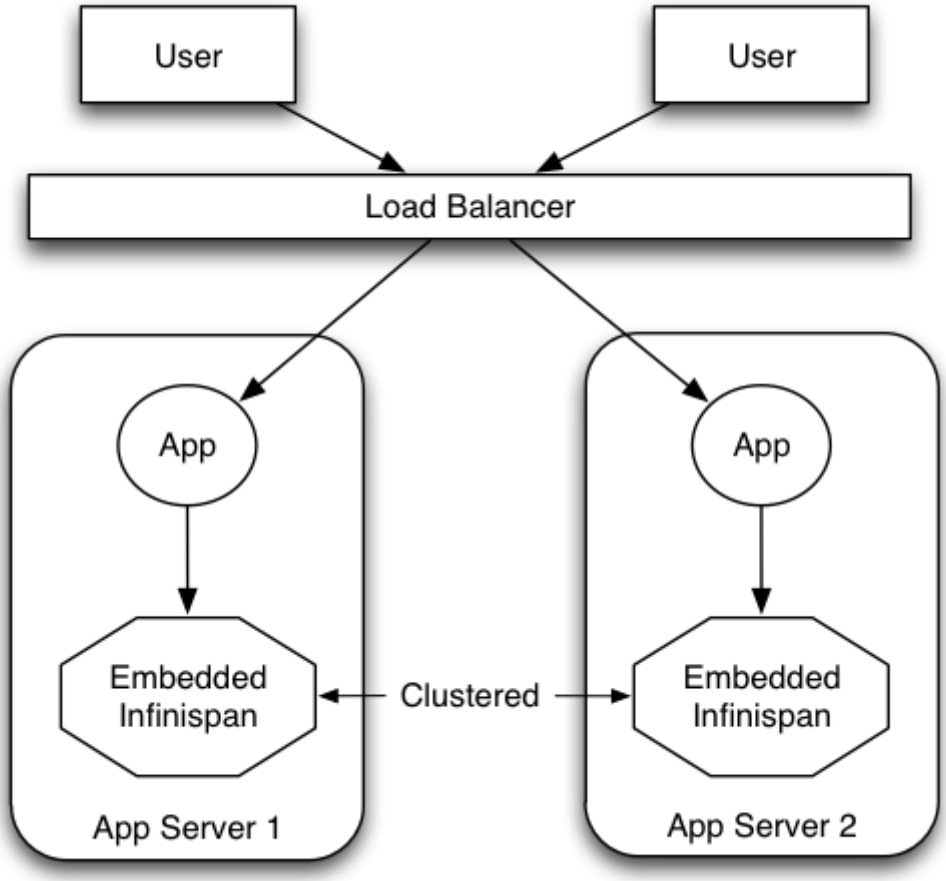

Figure 1. Peer-to-peer access

- Client-server mode: When applications access the data stored in a remote Infinispan server using some kind of network protocol.

3.1. Using the Client-Server Mode

There are situations when accessing Infinispan in a client-server mode that might make more sense than embedding it within your application. For example, this may apply when trying to access Infinispan from a non-JVM environment.

Since Infinispan is written in Java, if someone had a C\\ application that wanted to access it, it could not do it in using the p2p way. On the other hand, the client-server would be perfectly suited here assuming that a language neutral protocol was used and the corresponding client and server implementations were available.

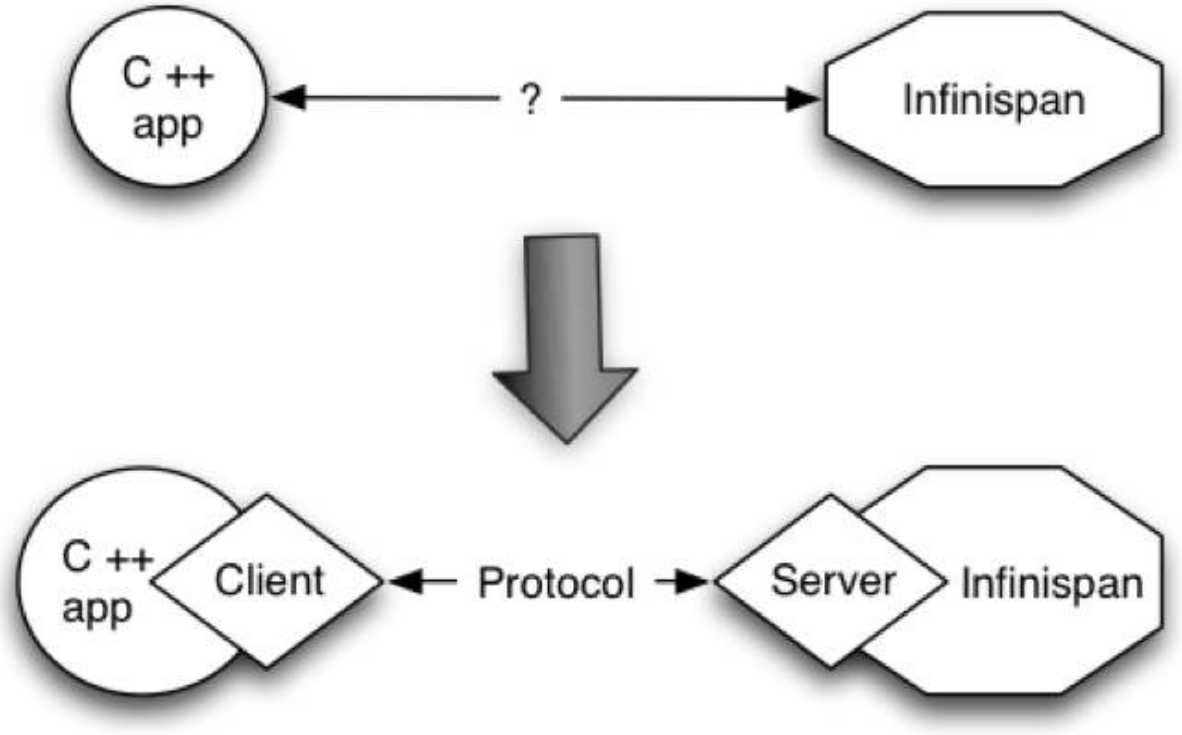

Figure 2. Non-JVM access

In other situations, Infinispan users want to have an elastic application tier where you start/stop business processing servers very regularly. Now, if users deployed Infinispan configured with distribution or state transfer, startup time could be greatly influenced by the shuffling around of data that happens in these situations. So in the following diagram, assuming Infinispan was deployed in p2p mode, the app in the second server could not access Infinispan until state transfer had completed.

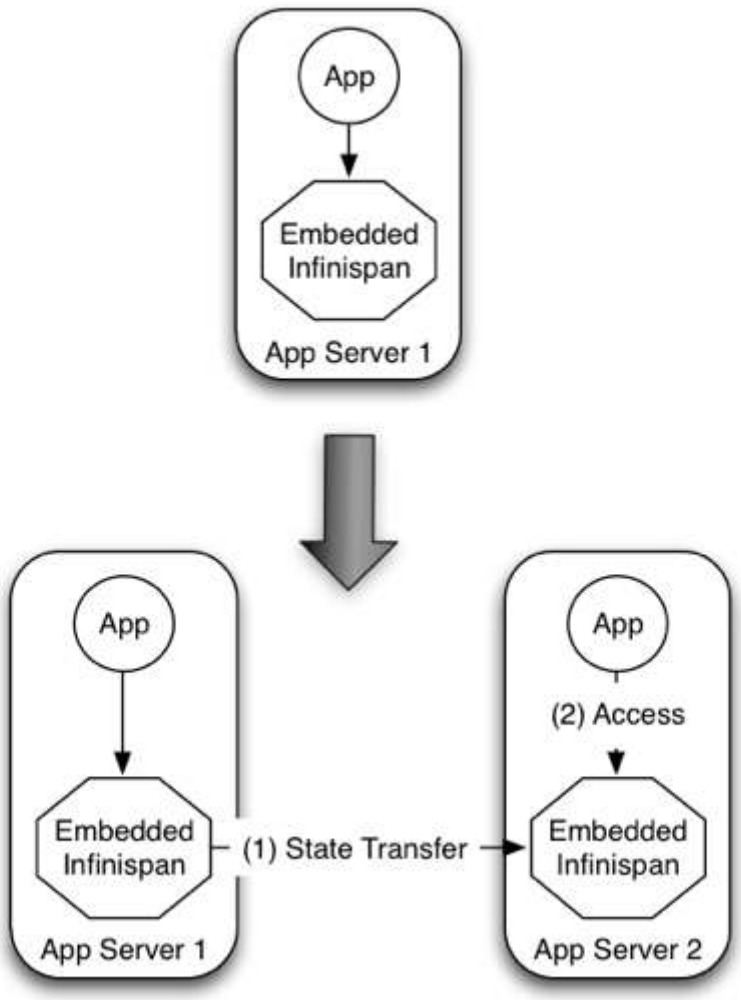

Figure 3. Elasticity issue with P2P

This effectively means that bringing up new application-tier servers is impacted by things like state transfer because applications cannot access Infinispan until these processes have finished. If the state being shifted around is large, this could take some time. This is undesirable in an elastic environment where you want quick application-tier server turnaround and predictable startup times. Problems like this can be solved by accessing Infinispan in a client-server mode because starting a new application-tier server is just a matter of starting a lightweight client that can connect to the backing data grid server. No need for rehashing or state transfer to occur and as a result server startup times can be more predictable which is very important for modern cloud-based deployments where elasticity in your application tier is important.

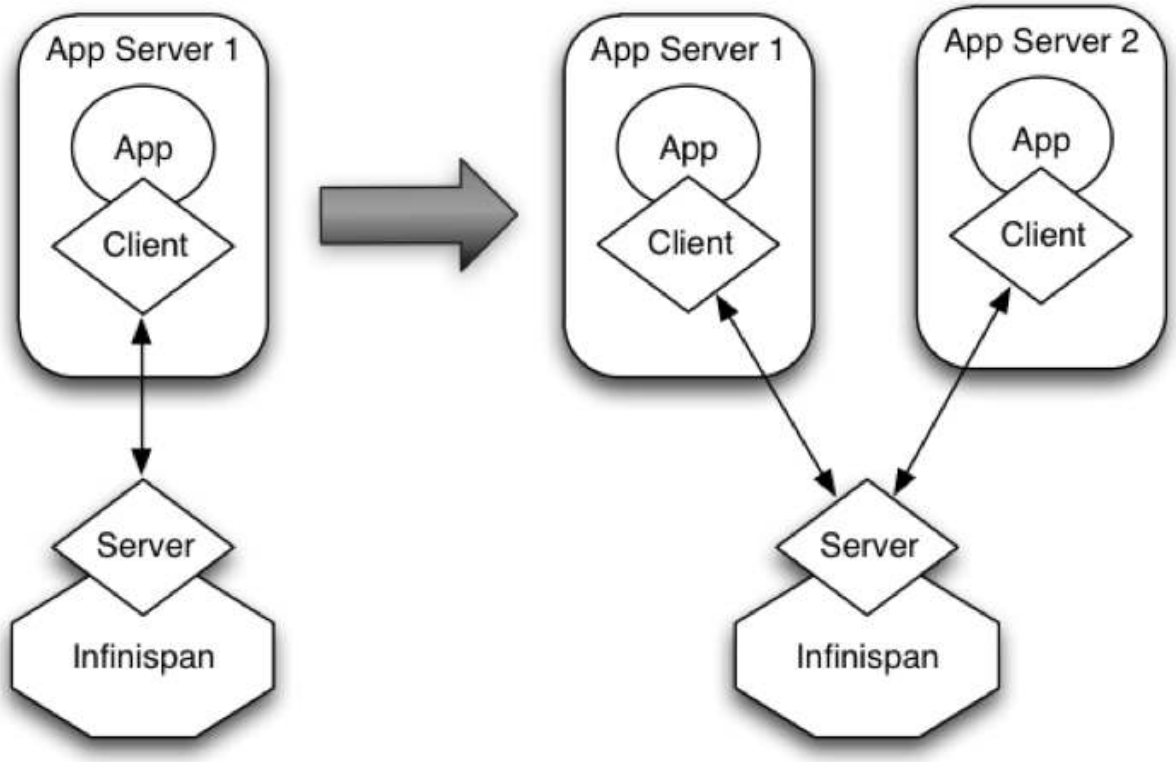

Figure 4. Achieving elasticity

It is common to find multiple applications needing access to data storage. Theoretically, you could deploy an Infinispan instance per each of those applications, but this could be wasteful and difficult to maintain. Consider databases; you do not deploy a database alongside each of your applications. Alternatively, you could deploy Infinispan in client-server mode keeping a pool of Infinispan data grid nodes acting as a shared storage tier for your applications.

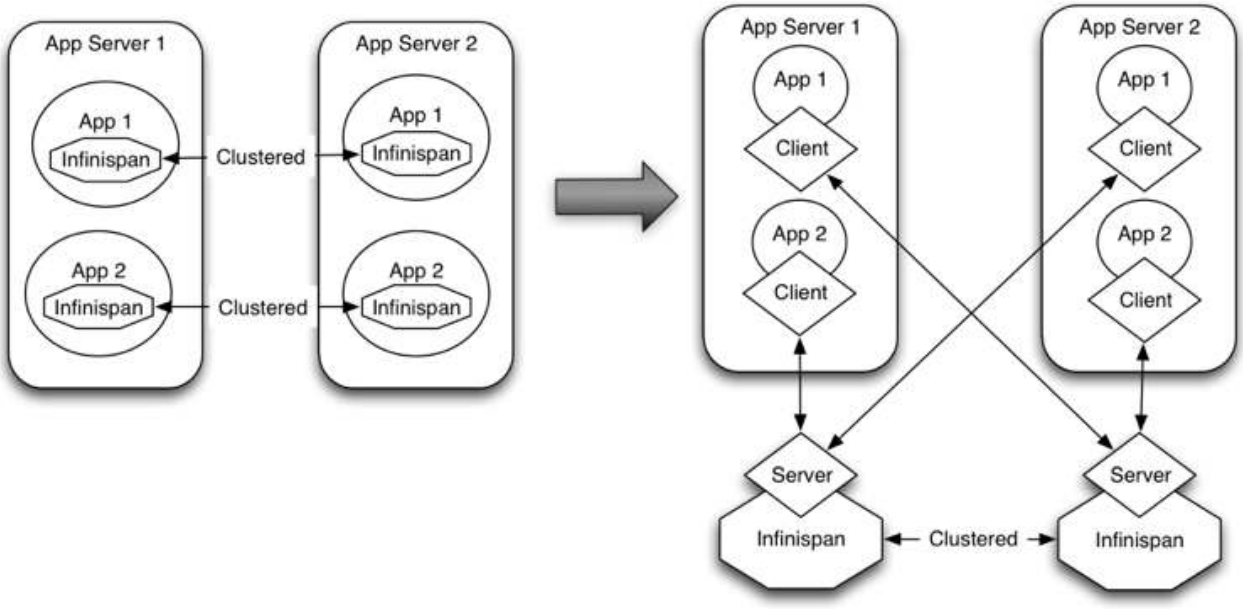

Figure 5. Shared data storage

Deploying Infinispan in this way also allows you to manage each tier independently. For example, you can upgrade r application or app server without bringing down your Infinispan data grid nodes.

3.2. Using the Embedded Mode

Before talking about individual Infinispan server modules, it's worth mentioning that in spite of all the benefits, client-server Infinispanstill has disadvantages over p2p. Firstly, p2p deployments are simpler than client-server ones because in p2p, all peers are equals to each other and this simplifies deployment. If this is the first time you are using Infinispan, p2p is likely to be easier for you to get going compared to client-server.

Client-server Infinispan requests are likely to take longer compared to p2p requests, due to the serialization and network cost in remote calls. So, this is an important factor to take in account when designing your application. For example, with replicated Infinispan caches, it might be more performant to have lightweight HTTP clients connecting to a server side application that accesses Infinispan in p2p mode, rather than having more heavyweight client side apps talking to Infinispan in client-server mode, particularly if data size handled is rather large. With distributed caches, the difference might not be so big because even in p2p deployments, you're not guaranteed to have all data available locally.

Environments where application tier elasticity is not important, or where server side applications access state-transfer-disabled, replicated Infinispan cache instances are amongst scenarios where Infinispan p2p deployments can be more suited than client-server ones.

Chapter 4. Frequently Asked Questions

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.

4.1. Project questions

4.1.1. 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.

4.1.2. 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.

4.1.3. 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.

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

Yes, and this is called Hibernate OGM .

4.1.5. 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.

4.1.6. 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.

4.2. Technical questions

4.2.1. 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.

4.2.2. 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.

4.2.3. 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.

4.2.4. 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).

4.2.5. 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.

4.2.6. 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)

Config / Mode of usage	Description
Replication	Configuring a clustered cache to replicate data asynchronously. 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.

4.2.7. 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.

4.2.8. 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

4.2.9. 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());
```

4.2.10. 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;
   }

   ...
}
```

Why am I getting a `StreamCorruptedException` during unmarshalling?

You may be modifying your object after inserting it in the cache. Infinispan assumes that once it was given an object (be it a key or a value), it can access that object on any thread, without any synchronization.

If you receive a `StreamCorruptedException` during unmarshalling, or a `ConcurrentModificationException` during marshalling, you should check that your application never modifies an object after passing it to `cache.put(key, value)` or after reading it from the cache with `cache.get(key)`.

The simplest fix is to configure the cache to store keys and values in binary form, e.g.

```
<infinispan>
  <cache-container>
    <distributed-cache name="myCache" mode="SYNC">
      <encoding media-type="application/x-protostream"/>
    </distributed-cache>
  </cache-container>
</infinispan>
```

The downside is that `cache.get(key)` is now more expensive, because it has to unmarshall the value every time. If the performance hit is not acceptable, you must instead modify your application to make a copy of the object in the cache and modify the copy, e.g.

```
Pojo value = cache.get(key);
Pojo modifiedValue = new Pojo(value);
modifiedValue.setProperty(newPropertyValue);
cache.put(key, modifiedValue);
```

4.2.11. 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).

4.2.12. 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.

4.2.13. 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.

4.2.14. 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.

4.2.15. 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}]
```

4.2.16. Clustering Transport questions

How do I retrieve the clustering physical address?

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

4.2.17. 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();
```

Chapter 5. Glossary

5.1. 2-phase commit

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

More resources

- Wikipedia article

5.2. 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

5.3. 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.

5.4. 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 .

5.5. 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

5.6. 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.

5.7. 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.

5.8. 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.

5.9. 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.

5.10. 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.

5.11. 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.

5.12. 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.

5.13. 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.

5.14. 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.

5.15. 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.

5.16. 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

5.17. 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.

5.18. 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.

5.19. 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.

5.20. 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.

5.21. 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.

5.22. 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.

5.23. 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.

5.24. 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.

5.25. 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.

5.26. Split brain

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

5.27. 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.

5.28. 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*.

5.29. 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.

5.30. 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*.

5.31. 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.