Infinispan is an extremely scalable, highly available key-value data store and data grid platform. In-memory data grids (IMDG) have recently been gaining a lot of attention, and the market for this technology is steadily increasing.

This book is a practical guide that covers all Infinispan features in a progressive manner with the help of real-world and ready-to-use examples. You will learn how to install Infinispan and create a perfect development environment. Next, you will learn how to persist data to a permanent store and integrate Infinispan with technologies such as Hibernate OGM, Amazon Web Services (AWS), and CDI. Finally, we will present a sample application that you can download from the JBoss Developer Framework. By the end of this book, you will be able to build extremely scalable applications, providing a highly available key/value data store.

Who this book is written for
This practical guide is intended for those who want to learn how to build extremely scalable applications. This book is easy to read and is aimed at Java enterprise developers with a solid knowledge of Java. However, no previous coding experience with Infinispan is required.

What you will learn from this book
- Install Infinispan and create the perfect development environment
- Get acquainted with topologies and strategies for different situations and data access patterns
- Familiarize yourself with scaling techniques and concepts including distributed transactions, ACID, and database sharding
- Monitor your Infinispan instances by using RHQ or JMX clients
- Control and manage transactions using JTA and use the available APIs to manipulate your cache data
- Create an application called TicketMonster and learn how to use it with Infinispan for reliable communication
- Learn about JGroups and how to use it with Infinispan for reliable communication
- Learn the concepts behind big data and how to work with Infinispan MapReduce API

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 7 'Understanding Transactions and Concurrency'
- A synopsis of the book’s content
- More information on Infinispan Data Grid Platform Definitive Guide

About the Author

**Wagner Roberto dos Santos** is an Italian Brazilian software architect living in Dublin, Ireland. He has more than 14 years of professional IT experience in software development, architecture, and Agile methodologies.

He is specialized in Java Enterprise technologies, and has deep knowledge of new emerging technologies such as cloud computing, NoSQL, and big data. As an architect, he supports teams in designing and developing scalable and maintainable solutions, following best practices, design patterns, and Domain-Driven Design practices.

As an agile coach, he supports teams in the adoption of Agile methodologies, such as Scrum, extreme programming, and Kanban, to make faster and better software deliveries to demanding time scales.

Until last year, Wagner was an MBA professor of distributed computing and Java Enterprise Development at FIAP and was also an instructor of several Agile and Java courses at Globalcode, a well-known Brazilian educational center specialized in software development.

Wagner contributes actively to the Agile and Java communities in Brazil. He is a member of SOUJava and speaks regularly at conferences such as The Developers Conference and Campus Party. He is also the former lead editor of the Architecture Queue at InfoQ Brasil and writes regularly for IT magazines, such as Mundo Java and Java Magazine.
Infinispan Data Grid Platform Definitive Guide

Relational database solutions have dominated the IT industry since the '80s. Over the last decade, relational databases have been losing its preference and importance for NoSQL databases and in-memory data grids.

There are many reasons involved in the decay of popularity, such as the fixed schema model, difficulties in scaling, and the centralized design that requires a heavy disk I/O for high write workloads.

Now, with the advent of big data and cloud computing technologies, enterprise systems have to deal with large data sets pushing its products to a new level, allowing businesses to reduce costs and distribute their data across multiple channels and applications.

Infinispan is an in-memory data grid and can help solve these challenges. It supports dynamic scalability and provides excellent performance for applications that require exposing and distributing data. Infinispan is an open source Java application that exposes a Cache interface that extends the java.util.Map interface, which can optionally distribute the cache data across a cluster.

This book will cover all aspects about Infinispan, from installation to tuning tips, and important concepts related to Distributed Transactions, Session Management, Map/Reduce, Topologies, and Data Access Patterns.
What This Book Covers

Chapter 1, Getting Started, introduces Infinispan and discusses the concept of data grid and its background. You will learn how to prepare the perfect environment to create an application using Infinispan and some valuable tips on how to contribute to the project.

Chapter 2, Barriers to Scaling Data, describes common approaches that are used to overcome obstacles such as performance, scalability, and high availability. It also talks about how these solutions can be improved using Infinispan and why the reader should use Infinispan and its benefits.

Chapter 3, Using the APIs, covers the core aspects of Infinispan. You will learn the basic concepts of the Infinispan API, such as how to use the core API to store, retrieve, and remove data from it. It also covers configuration.

Chapter 4, Infinispan Topologies, covers various cache topologies supported by Infinispan and provides guidance on when to use each one and how to configure them.

Chapter 5, Data Access Patterns, presents some caching access patterns and their advantages and disadvantages.

Chapter 6, Case Study – The TicketMonster Application, introduces the Ticket Monster application to show how to put into practice the strategies and concepts you learned in the previous chapters.

Chapter 7, Understanding Transactions and Concurrency, introduces important concepts about transactions and how you work with transactions with Infinispan.

Chapter 8, Managing and Monitoring Infinispan, dives into management and monitoring Infinispan details. It shows you how to enable JMX to collect runtime information and how to manage multiple Infinispan instances spread across different servers using RHQ.

Chapter 9, Server Modules, teaches you about Infinispan modules, which allow you to access an Infinispan data grid from remote clients and from platforms and languages other than Java.

Chapter 10, Getting Started with Hibernate OGM, discusses Hibernate Object/Grid Mapper (OGM) and presents the Hibernate OGM architecture, its current features, and how to integrate Hibernate OGM with Infinispan.

Chapter 11, An Introduction to JGroups, shows that Infinispan uses JGroups as a network transport. In this chapter, we will see more details on JGroups and how it relates to Infinispan.

Chapter 12, Advanced Topics, discusses Infinispan advanced topics, such as cross-site replication, CDI, the Infinispan MapReduce framework, and how to plug Infinispan with user defined externalizers.
In the previous chapter, we introduced the Ticket Monster application to learn about JBoss Developer Framework and JBoss-related technology. We added some requirements in order to analyze how Infinispan can be used to improve scalability.

In this chapter, we will talk about transactions and how you can work with transactions with Infinispan. We'll discuss the following topics:

- Transaction fundamentals
- Java Transaction API
- Transaction models
- Locking and concurrency control

Designing your cache application and the transaction configuration that connects your application to a data store for optimal performance isn't easy. This chapter will help you identify the best concurrency model and help you determine how to better manage your transactions.

To begin with, we are going to be cover the basic concepts required to understand how Infinispan is structured and how applications can use it.
Transaction fundamentals

Before we start to learn how Infinispan deals with transactions, let's learn the basics about transactions in order to extract the best from this feature.

By definition, a transaction allows you to set the boundaries of a user-defined series of logically related read or write operations (get() or put()). All changes brought by a write operation (update, remove, or add an entry in the cache) are either undone or made permanent at the same time.

The processing of these transactions is divided into individual, atomic operations and might help you isolate one transaction from another transaction in a multiuser application. Each transaction at the end must complete the processing with success or failure as a complete unit, and a transaction should not complete the processing in an intermediate state.

To execute all your data grid operations inside a transaction, you have to mark the boundaries of that transaction. You must start the transaction and at some point commit the changes. Generally, you conclude a transaction with the COMMIT or ROLLBACK statements, to ensure that all or none of your changes are stored in the grid.

The commit processes writes the data into the cache and makes it visible to other users. Rolling back discards any changes that have been made to any cache entry since the beginning of the current transaction.

Let's exemplify this concept with the classical example of a bank transfer operation, in which a customer performs a transfer of $50,000 between accounts. The bank application needs to perform an operation that consists of three separate but related actions, as follows:

1. Debit the source account by the required amount of $50,000.
2. Credit the target account by the required amount of $50,000.
3. Write a log entry recording the transfer.

When viewed as three separate operations, it's not difficult to imagine a disaster. These operations should be performed as a unit. Imagine that for some reason the debit operation succeeds, and the bank's central server fails just after the debit operation, before the credit operation completes. This kind of error can cause a great deal of damage to the bank's image.

It is important to understand that transactions in Infinispan are not like transactions on a relational database product. Infinispan is an in-memory database, it utilizes volatile storage for in-memory data, which means that if the application goes down all the data cached in the memory is lost.
It's the responsibility of the cache to recreate data upon startup, either from backup nodes or from other persistent storage systems, if you have a cache loader/store configured.

Another important responsibility of the data grid is to protect your information and to guarantee that the data between different grid nodes must remain consistent, which is part of the ACID criteria commonly applied to transactional database management systems.

In a distributed environment, things get more complicated. Infinispan uses a two-phase commit (also known as 2PC) to manage all the cluster nodes that are participating in a distributed transaction. A well known practice to ensure the atomicity of a transaction that accesses multiple resource managers.

We have covered about ACID and two-phase commit in Chapter 2, Barriers to Scaling Data.

### Java Transaction API

Infinispan can be prepared to participate in JTA-compliant transactions.

JTA or the Java Transaction API, allows developers to demarcate transactions in a way that makes your code independent of the implementation details of the transaction manager.

With JTA, developers have to worry about getting their code up and running on the platform and using code that manages transactions rather than connections, such as JDBC. JTA exposes several interfaces such as `javax.transaction.UserTransaction`, where the developer can start the transaction with the `begin()` method and either the `commit()` or `rollback()` method to terminate the transaction and the `javax.transaction.TransactionManager` API to participate in the transaction lifecycle.

One of the responsibilities of the transaction manager when a transaction is about to be committed is to ensure that everything is committed or rolled back.

When more than one Infinispan instance is involved in the transaction, the management of the commit gets more complicated, an XA protocol with the two-phase commit is used. A basic difference with a normal distributed transaction is that an XA protocol includes an additional prepare phase that occurs just before the actual commit phase.
In computing, the XA (eXtended Architecture) standard is a specification published by the Open Group for Distributed Transaction Processing. The goal of XA is to allow multiple resources to be accessed in the same transaction, preserving the ACID properties across applications.

The JTA specification provides the XAResource interface, a Java mapping of the Open Group standard XA interface.

Before asking any of the instances to commit the changes, the transaction manager must first check with all the participants of the grid to prepare to commit. When one of the participant instances acknowledges that it is ready (or prepared) to commit the transaction, it is an indication that it can commit the transaction. If any of the participants fails to prepare, the transaction manager will rollback the whole transaction.

If all participants confirm the first phase (prepare), in the second phase, the transaction manager will ask all the resources to commit the transaction, which cannot fail otherwise, as we said, the transaction will be rolled back.

The transaction manager can coordinate a transaction that spans several resources.

Infinispan will perform the following operation for every transaction:

- Get the current transaction associated with the current thread
- If not already done, the transaction manager obtains and registers the XAResource with the transaction to receive notifications when a transaction is committing or is rolled back.

To participate in a distributed transaction, you must enlist the XAResource instance with the Transaction object.

A TransactionManager object can be obtained from an AdvancedCache class using the getTransactionManager() method, which is as follows;

```java
TransactionManager tm = cache.getAdvancedCache().getTransactionManager();
```

However, before you obtain a TransactionManager object, you have to specify the settings for a transaction either declaratively or programmatically. In either case, the result is the same.

If you want to use a configuration file, you can define the transactional characteristics of the cache through the <transaction> tag, under a cache configuration for a specific cache.
In Infinispan 6, you can include the <transaction> tag under a <default> tag in order to define default transactional behavior to all caches belonging to the cache manager.

The following example shows a simple transaction configuration. We are going to explain all aspects of these sample configurations in this chapter.

The code in Infinispan 6 is as follows:

```xml
<transaction transactionManagerLookupClass="org.infinispan.transaction.lookup.GenericTransactionManagerLookup" cacheStopTimeout="30000" use1PcForAutoCommitTransactions="false" autoCommit="true" useSynchronization="false" transactionMode="TRANSACTIONAL"/>
```

This code is written in Infinispan 7, as follows:

```xml
<local-cache name="pessimisticCache">
  <transaction transaction-manager-lookup="org.infinispan.transaction.lookup.GenericTransactionManagerLookup"
    stop-timeout="30000"
    auto-commit="true"
    locking="PESSIMISTIC"
    mode="NON_DURABLE_XA" />
</local-cache>
```

Otherwise, you can configure a transactional cache programmatically as well:

```java
Configuration config = new ConfigurationBuilder()
    .transaction()
    .transactionMode(TransactionMode.TRANSACTIONAL)
    .transactionManagerLookup(new GenericTransactionManagerLookup())
    .cacheStopTimeout(30000l)
    .use1PcForAutoCommitTransactions(false)
    .autoCommit(true)
    .useSynchronization(false)
    .build();
```

The cacheStopTimeOut / stop-timeout attribute defines the amount of time the TransactionManager object is allowed to wait for any local or remote ongoing transaction to finish their job after a cache is stopped.
Another configuration option that we have used is the Synchronization, a mechanism that allows Infinispan to be notified before and after a transaction completes during the two-phase commit process.

By default, Infinispan registers itself as a participant in distributed transactions as a full XAResource. But, there can be some situations like when you use Infinispan as a second level cache in Hibernate, where it is not required to be a fullXA resource in a transaction.

In these cases, you can register synchronization with the TransactionManager object by enabling transaction enlistment.

Using the Infinispan 6 configuration, you can enable the transaction enlistment using the useSynchronization attribute. To enable transaction enlistment using the Infinispan 7 declarative configuration, you’ll have to set the transaction mode to NON_XA, as you can see in the following example:

```xml
<transaction mode="NON_XA" />
```

In order to look up a native transaction manager within your environment, Infinispan makes use of an implementation of the TransactionManagerLookup interface. The transactionManagerLookupClass/transaction-manager-lookup attribute configures the Transaction manager lookup directly using an instance of TransactionManagerLookup.

During the startup phase of the cache, it will create an instance of the TransactionManagerLookup interface and invoke the getTransactionManager() method, responsible for a reference to the JTA TransactionManager.

Currently, Infinispan provides four transactional manager lookup classes, as follows:

- **DummyTransactionManagerLookup**: A TransactionManagerLookup interface for testing, it is not recommended for use in the production environment and it has important limitations related to concurrent transactions and recovery.

- **JBossStandaloneJTMangerLookup**: This is the recommended lookup class used in a standalone environment and it overcomes all limitations and deficiencies of the DummyTransactionManagerLookup interface.

- **GenericTransactionManagerLookup**: This is the lookup class that tries to find the transaction manager in the most popular application servers. If the TransactionManager object is not found, it will use a DummyTransactionManagerLookup interface.
• **JBossTransactionManagerLookup**: If you are using Infinispan with JBoss Application Server, then this is your best option; it locates the transaction manager running within a JBoss AS instance.

Most cache operations within the scope of a JTA Transaction will propagate a runtime CacheException or any of its subclasses on failure, causing the transaction to be automatically marked for rollback for all application exceptions.

### Transactional modes

The transaction mode attribute is slightly different in Infinispan 6 and 7. In Infinispan 6, the `transactionalMode` attribute configures whether the cache is transactional or not.

In the next example, we can see a sample configuration using Infinispan 6:

- `<transaction transactionMode="NON_TRANSACTIONAL"/>
- `<transaction transactionMode="TRANSACTIONAL"/>

Starting with Infinispan 5.1, you cannot mix transactional and non-transactional operations anymore; you have to decide which cache mode you want to use.

There are many reasons to follow this path, but one of the most important reasons is the clean semantics on how concurrency is managed between multiple requestors for the same cache entry. If we were allowed to mix transactional and non-transactional operations, we would run the risk of experiencing unexpected behavior, such as deadlocks, interacting with a transactional and non-transactional code.

In Infinispan 7, we have the mode attribute, which configures the transaction type for the cache to one of the following modes: `NONE`, `BATCH`, `NON_XA`, `NON_DURABLE_XA`, and `FULL_XA`.

The code in Infinispan 7 is as follows:

```xml
<transaction transaction-manager-lookup="org.infinispan.transaction.lookup.GenericTransactionManagerLookup" mode="NON_XA"/>
```

Alternatively, you can configure a transactional cache programmatically:

```java
Configuration c = new ConfigurationBuilder().transaction().transactionMode
(TransactionMode.TRANSACTIONAL).build();
```

If unspecified, it defaults to `NON_TRANSACTIONAL`.
Non-transactional data access

Infinispan provides an autocommit mode on every new data grid access, by default. The autocommit mode is useful for an interactive data access, when you have a sequence of operations that you do not consider to be part of a transaction, for example, when you connect to your cache to get some values, and maybe even remove or add entries. The default autocommit mode on the cache is perfect for these scenarios—after all, you will probably not want to begin and end a transaction for every statement you write and perform in the grid.

Using the autocommit mode, a short transaction begins and ends for each operation you send to the grid. You're effectively non-transactional, because in this mode Infinispan cannot guarantee the atomicity or isolation properties for your session.

On a transactional cache with auto commit enabled, any call performed outside a transaction's scope is transparently wrapped within a transaction. Before the call, Infinispan adds the logic for starting a transaction and performs a commit after the call.

Transactional models

A transactional cache in Infinispan supports two different transactional models, optimistic and pessimistic.

The optimistic model refers to an approach in which transactions are allowed to proceed, with conflicts resolved as late as possible, deferring lock acquisitions for the transaction, in order to prepare time.. The entry will not be immediately locked when it is accessed by a transaction, which means that the cache entries will be available to other transactions for concurrent access, opening up the possibility of conflicts.
At commit time, when the entry is about to be updated in the grid, Infinispan will compare the version of the current object to the version that was initially saved at the moment the entry was first requested in the transaction. If both versions differ from each other, Infinispan will consider that a conflict exists, and will mark the transaction for rollback. This avoids deadlocks, optimizes the lock acquisition time, and increases throughput significantly.

On the other hand, the pessimistic model refers to an approach in which potential conflicts are detected and resolved earlier. Cluster wide locks are acquired for every write operation and only released after the transaction commits.

**Optimistic transaction**

As we said, during optimistic transactions, locks are acquired during the prepare phase and are held until the time the transaction commits (or rollbacks).

Optimistic transactions are recommended when the probability of two different users change the same data in parallel is low.

The following diagram shows how the optimistic lock works:

![Optimistic Lock Diagram](image-url)
Understanding Transactions and Concurrency

The diagram shows the users, John and Alice, issuing a put operation to the k1 object in a different transaction context. We can see that John invoked the put() method first, but Alice performed a commit operation in her transaction (Transaction 02) before John saved his changes.

After invoking the commit operation, the Transaction Manager starts the prepare phase and locks the k1 object, and at the end of the commit phase, it releases the lock, making the k1 object stale to John. When he tries to commit his transaction, Infinispan identifies a conflict, throws an exception, and marks the transaction for rollback.

In Infinispan 6, optimistic transactions can be enabled specifying the lockingMode attribute in the configuration file, as you can see in the next example:

```xml
<namedCache name="transactionalOptimistic">
  <transaction transactionMode="TRANSACTIONAL" lockingMode="OPTIMISTIC"/>
</namedCache>
```

You can enable optimistic transactions, specifying the locking attribute within the <transaction> element. Check the following example for the Infinispan 7 configuration that creates a cache with an optimistic locking schema. The code in Infinispan 7 is as follows:

```xml
<local-cache name="transactionCache">
  <transaction transaction-manager-lookup=
                "org.infinispan.transaction.lookup.
                JBossStandaloneJTAManagerLookup"
               mode="NON_XA" locking="OPTIMISTIC" />
</local-cache>
```

Otherwise, it can be programmatically configured as follows:

```java
Configuration c = new ConfigurationBuilder().transaction().
        lockingMode(LockingMode.OPTIMISTIC).build();

assert c.transaction().lockingMode() == LockingMode.OPTIMISTIC;
```
Pessimistic transaction

Pessimistic transactions prevent that other concurrent transactions modifying the same entry. Infinispan obtains locks on entry keys at the time it is written. The following diagram shows how the pessimistic lock works:

In our example, when the `cache.put(k1,v1)` method returns, the `k1` object will be automatically locked in the transaction, preventing concurrent transactions from updating it. Concurrent transactions are allowed only to read the `k1` object, but not update it, which is the case with Alice's transaction. The lock is released when the transaction completes via a commit or rollback operation.

You can enable or disable pessimistic transactions in the configuration file by changing the correspondent locking attribute.

The following sample code shows how to configure pessimistic locking with Infinispan 6:

```xml
<namedCache name="transactional"/>
<transaction transactionMode="TRANSACTIONAL"
  lockingMode="PESSIMISTIC"/>
</namedCache>
```
Understanding Transactions and Concurrency

Configure pessimistic locking with Infinispan 7 as follows:

```xml
<local-cache name="pessimisticCache">
  <transaction transaction-manager-lookup="org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup" locking="PESSIMISTIC"
      mode="NON_DURABLE_XA"/>
</local-cache>
```

Otherwise, it can be programmatically configured as follows:

```java
Configuration c = new ConfigurationBuilder().transaction().lockingMode(LockingMode.PESSIMISTIC).build();
assert c.transaction().lockingMode() == LockingMode.PESSIMISTIC;
```

Choosing the better transaction model

As we saw, the pessimistic model will fail when it faces a conflict; while the optimistic model will give an opportunity to resolve the conflict at commit time.

In other words, optimistic transactions do not lock cache entries, two different transactions might change the same entry at the same time, and an eventual conflict will be detected during the second phase, if a second transaction attempts to flush or commit locked entry. This happens because optimistic transactions will rollback the transaction if the data has changed after the application reads the data, and before the application commits the transaction (writeSkewCheck).

Unlike optimistic transactions, pessimistic transactions generally lock the entry they act on, preventing other concurrent transactions from using it, which avoids conflicts between transactions, so maybe the pessimistic model is better for applications that have a high risk of contention. The disadvantage of pessimistic transactions is that they consume additional resources since the resource is locked for each write operation.

In most scenarios, optimistic transactions are more efficient and can be the best choice for most applications, since they offer higher performance, better scalability, and might reduce the risk of a deadlock.

Batch mode

Infinispan provides several methods of putting data in the cache, such as the standard map operations such as cache.put(...), cache.putAll(...), or cache.putIfAbsent(...) an overloaded form of ConcurrentMap.putIfAbsent(), which only stores the value if no value is stored under the same key.
However, these methods will result in a separate network call for each operation, which is not suitable for scenarios where large amounts of data must be loaded into the data store, especially for caches in replication mode. This is the case, for instance, building a mirror site or importing data to the cache when transaction control is not important.

For these cases, Infinispan provides the ability to batch multiple cache operations through the interface org.infinispan.commons.api.BatchingCache that provides the startBatch() method to start the batch process, and endBatch(boolean) to complete the process.

The Infinispan batching mode allows atomicity and other transactional characteristics, but doesn't provide full JTA or XA capabilities.

In the batching mode, all configuration options related to the transaction such as syncRollbackPhase, syncCommitPhase, useEagerLocking, and eagerLockSingleNode are applied as well. Internally, the batching process starts a JTA transaction using a simple internal TransactionManager implementation without recovery. And all the entries in that scope will be queued on the same instance and changes are batched together around the cluster in a part of the completion process, reducing replication overhead for each update in the batch.

When you use the batch mode, there is no transaction manager defined.

By default, invocation batching is disabled; how to enable batch process declaratively is shown in the following example. Invocation batching configuration with Infinispan 6 can be done as follows:

```xml
<invocationBatching enabled="true" />
```

For Infinispan 7, we don't have more `<invocationBatching>` elements, so to enable invocation batching you have to specify the transaction mode to BATCH:

```xml
<transaction transaction-manager-lookup="org.infinispan.transaction.lookup.
  JBossStandaloneJTAManagerLookup" mode="BATCH"
  locking="OPTIMISTIC"/>
```

Otherwise, it can be programmatically configured, on the InvocationBatchingConfiguration object:

```java
conf.invocationBatching().setInvocationBatchingEnabled(true);
```
After this, configure your cache to use batching. Perhaps the easiest way to illustrate this is to demonstrate a simple scenario showing how to import a CSV file into Infinispan, in order to prepopulate a cache before the application makes use of it.

CSV stands for Comma Separated Values.

First, we created a CSV file in the resource folder with the name csv_guest_list.csv, with the following content:

```
ID, first_name, last_name, document_number, birth_date
1, John, Wayne, 832218, 19801112
2, Eddy, Murphy, 822712, 19901003
3, Fred, Mercury, 872211, 19640321
4, Juliette, Lewis, 862211, 19720804
5, Kate, Moss, 872911, 19790413
```

The content of the CSV file is a guest list for a given event. Next, we created a POJO class Guest for the imported data and a utility class GuestListImporter to import CSV files.

Finally, we can use the batching process by calling startBatch() and endBatch(), as highlighted in the following example:

```java
Cache<Integer, Guest> cache =
    container.getCache("batchingCacheWithEvictionAndPassivation");
List<Guest> guests = new
    GuestListImporter().parseGuestFile("guest_list.csv");
try{
    cache.startBatch();
    for(Guest guest : guests){
        // do some processing
        cache.put(guest.getId(), guest);
    }
    assertEquals(guests.size(), 5);
    cache.endBatch(true);
} catch(Exception ex){
    cache.endBatch(false);
}
```

Note that the endBatch() method receives a Boolean parameter, which completes the batch if true; otherwise, it will discard all changes made in the batch.
Transaction recovery

Although XA transactions possess the ACID characteristics in order to guarantee the atomicity of operations, our system must be able to handle failures in order to guarantee the consistency of customer data, which can occur at any time due to unexpected server crash or network loss.

To guarantee transaction consistency, Infinispan supports transaction recovery, a well known feature of XA transactions, present in the specification published by the Open Group.

Let's suppose a situation where a customer buys from Ticket Monster, a ticket for a specific show, but we have to save the ticket in two different nodes in Infinispan. The Transaction Manager will be responsible for communicating with both resources that are in use.

When the transaction manager commits, in phase one, the transaction manager asks both resources to prepare the commit. Then both resources verifies they can persist the data and each resource sends an acknowledgement to the coordinator. In the second phase, when resources are requested to commit, for some reason one of the them fails to complete the commit operation, thus leaving the cache data in an inconsistent state.

In situations like this, Infinispan supports automatic transaction recovery coordinated by the Transaction Manager, to make sure data in both resources ends up being consistent.

The Transaction Manager works with Infinispan to determine any transaction in an in-doubt state that was prepared but not committed. If there are no left-pending transactions, it will proceed normally, otherwise, the Transaction Manager will request the Infinispan cluster to complete the commit or force the rollback to release any resource.

There are some cases where Infinispan will not be able to recover all transactions in an in-doubt state, and where recovery could not be complete, for these cases, Infinispan can list transactions that require a manual intervention.

As a system administrator, you can configure Infinispan to receive notifications about these cases that require manual intervention by e-mail or log alerts, which require some configuration on the transaction manager.
Infinispan provides JMX tooling to view any transactions that require manual intervention, which we will see in Chapter 8, Managing and Monitoring Infinispan.

The following diagram shows a graphical illustration of the concept, with a node failure in the originator:

From the image you can see that the changed data is held by the Recovery Manager only for in-doubt transactions, being removed for successfully completed transactions after the commit or rollback phase is complete.

You can enable transaction recovery per cache level through XML configuration. If you are using Infinispan 6, you can enable transaction recovery by adding a `<recovery>` element, a child element of the `<transaction>` parent element, as follows:

```xml
<transaction transactionManagerLookupClass="org.infinispan.transaction.lookup.GenericTransactionManagerLookup" transactionMode="TRANSACTIONAL">
    <recovery/>
</transaction>
```
For recovery to work, the `useSynchronization` attribute must be set to `false`.

Enabling transaction recovery with Infinispan 7 is considerably easier than its predecessor; for the recovery to work, you must set the mode attribute to `FULL_XA`, because it only works if the transaction is registered as a full XA resource:

```xml
<replicated-cache name="transactionCacheWithRecoveryExample" mode="SYNC">
  <transaction transaction-manager-lookup="org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup" mode="FULL_XA" recovery-cache="recoveryCache"/>
</replicated-cache>
<local-cache name="recoveryCache"/>
```

Alternatively, you can enable recovery programmatically on the `RecoveryConfiguration` class, as follows:

```java
config = new ConfigurationBuilder()
  .transaction()
  .transactionMode(TransactionMode.TRANSACTIONAL)
  .transactionManagerLookup(new GenericTransactionManagerLookup())
  .recovery()
  .enable()
  .recoveryInfoCacheName("recoveryCache")
  .build();

// or just check its status
boolean isRecoveryEnabled = config.isTransactionRecoveryEnabled();
```

In Infinispan 6, the `<recovery>` element defines the attributes enabled that you use to enable transaction recovery.

The `recoveryInfoCacheName` class in Infinispan 6 (`recovery-cache` in Infinispan 7) can be optionally set up to provide a name to the cache that will hold all recovery information, it's not mandatory, but if omitted, the cache's default name is `__recoveryInfoCacheName__`. 
Understanding Transactions and Concurrency

All in-doubt transaction data will be backed up at a local cache specified through the recoveryInfoCacheName configuration attribute, if available, which allows data to be evicted to the disk through the cache loader as normal cache, in case it gets too big.

**Integrating with Transaction Manager**

During recovery, the Transaction Manager communicates to all the resource managers that are participating in the transaction. But there can be some situations where we may need to run the recovery in a different process from the one running the transaction.

To retrieve the transactions that are currently in a prepared state, the Transaction Manager uses the XAResource.recover method on it. To obtain a reference to an Infinispan XAResource, the following API can be used to return the XAResource associated with this cache:

```java
XAResource xar = cache.getAdvancedCache().getXAResource();
```

Besides the fact that the XA specification allows to run the recovery in a different process, today in Infinispan it's only possible to run the recovery process in the same process, where the Infinispan instance exists.

A future release is planned for Hot Rod clients to support transactions. We will see Hot Rod in action in Chapter 9, Server Modules.

**Locking and concurrency control**

As we have seen, we can have one or more clients participating on a transaction that issues read and write operations to a cache. When transactions execute concurrently on the same cache entry, the interleaved execution of their reads and writes by the cache can produce undesirable results. On the other hand, a good cache solution must guarantee consistency of data while allowing multiple transactions to read/write data concurrently.

Concurrency control is an activity that is important to avoid unexpected results; its primary goal is to ensure that all transactions will have the same effect as a serial one.

Traditionally, the problem of concurrency is solved providing transaction isolation, keeping a single version of the data and locking other requests to manage concurrency.

Locking is essential to avoid change collisions resulting from simultaneous updates to the same cache entry by two or more concurrent users.
If locking is not available and several users access a distributed cache concurrently, concurrency problems may occur if their transactions access the same data at the same time.

Concurrency problems include:

- **Dirty Reads**: This means that a client can read uncommitted changes made by another client, which is shown in the following figure:

- **Non-Repeatable Reads**: A client gets a cache entry, another client changes the same cache entry and commits, and the first connection re-reads the same entry and gets the new result, which is shown in the following figure:
• **Phantom Reads**: It may happen when a client performs an insert or delete operation against an entry that belongs to a collection of entries being read by another transaction. Then the second client re-reads using the same condition and gets the new entry, which is shown in the following figure:

• **Lost Updates**: It may happen when two or more clients, during their individual transactions select the same entry and change their value. The transactions are independent and unaware of each other; lost updates might happen when the last update to the entry overwrites updates performed by other clients, which results in lost data, as shown in the following figure:
To overcome these phenomena, the ANSI/ISO SQL standard defines four levels of transaction isolation that will provide a different level of access control which may or may not be more restrictive.

As you can deduce by the ANSI/ISO standard, isolation levels have their origins in relational databases. The defined transaction isolation levels are:

- **READ COMMITTED**: This level means that the read operation can see only data committed before the operation began, because the read lock is released immediately after operation. But, during the write operation, the system keeps the lock until the transaction commits. Higher concurrency is possible when using this level. This mode prevents dirty reads but allows problems of lost updates, nonrepeatable reads, and phantom reads to occur.

- **READ UNCOMMITTED**: This level means that transaction isolation is disabled. This mode allows all aforementioned concurrency problems.

- **REPEATABLE READ**: This level means that every lock acquired during a transaction is held until the end of the transaction. This mode prevents the problem of non-repeatable reads, because once data has been written, no other transaction can read it, but it allows problems related to phantom reads to occur.

- **SERIALIZABLE**: This is the most restrictive isolation level. This level requires read and write locks to be released at the end of the transaction and does not allow the insertion of new entries into the range that is locked. This mode prevents all concurrency problems, but it can cause serialization failures.

Please consult the java.sql.Connection API for more references.

As we have seen before, Infinispan uses the two-phase commit (2PC) protocol, in order to coordinate with all the processes that participate in a distributed transaction. However, there are also costs related to 2PC in a replicated cache. First, the cost associated to memory consumption by the replicated entries that can be reduced drastically by changing the cache mode to distribution. Also, the more objects you have in your data grid, the more and more expensive it becomes to ensure the consistency of these objects.

You might be saying that the distribution clustering mode can overcome these disadvantages, by configuring the Infinispan cache in such a way that each item is replicated in a limited number of nodes. But, in the distribution mode, we have an additional overhead associated with the coordination costs.
For these reasons, Infinispan has opted for relaxing consistency, ensuring weaker semantics in order to allow more efficient implementations.

Specifically, Infinispan supports the following isolation levels: read committed and Repeatable read. But remember, transactions in Infinispan are not like transactions in a relational database product.

In Infinispan, READ COMMITTED and REPEATABLE READ work slightly differently than databases. In READ COMMITTED, reads can happen anytime, while in REPEATABLE READ, once the data has been written to the cache, no other transaction can read it so there's no chance of later re-reading the data under the same key and getting back a completely different value.

Infinispan has only been able to provide all of this thanks to Multi-Version Concurrency Control (MVCC), which is the subject of the next section.

Multiversion concurrency control
Infinispan implements a concurrency schema well known in relational databases called multi-version concurrency control (MVCC).

MVCC offers many advantages over Java synchronization and JDK locks for access to concurrent data. Infinispan can achieve an excellent performance by maintaining multiple versions of data instead of using any synchronization or locking for reader threads.

It allows readers to get consistent data without blocking writers. And 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, improving the general performance of cache applications in a multuser environment.

The Infinispan implementation of MVCC is well optimized for read operations, because for read operations, Infinispan does not acquire an explicit lock on the data being read. On the other had, to ensure one concurrent writer per entry, Infinispan acquires a lock for write operations, queuing up the changes to the entry.

Configuring isolation levels in Infinispan
You can configure the isolation levels by adding a `<locking>` configuration element, which lets you define the local, concurrency level, isolation level, lock acquisition timeout, and other characteristics of the cache.
Isolation levels define the level a reader can see of a concurrent write. Depending on the isolation level you choose, you will have a different behavior in how the state is committed back to the cache.

The default isolation level used by Infinispan is **READ_COMMITTED**, which also performs the best, and is generally good enough for most applications.

Let's take a look at a more detailed example that shows the difference between **READ_COMMITTED** and **REPEATABLE_READ**. With the read committed isolation level, during two consecutive read operations on the same key, if the key is updated by a third transaction, the second read will return the new value, which is shown in the following figure:

As you can see, in this diagram we are showing an example of a possible scenario with the READ COMMITTED isolation level, and at the end of the diagram, in step 7 the second read returns a new value \( v_2 \).
However, if we were using the `REPEATABLE_READ` isolation level, step 7 would still return \( v1 \). So, if you want to read the same entry multiple times within a transaction, we recommend you to use `REPEATABLE_READ`. The `REPEATABLE_READ` isolation level also allows for an additional safety check known as Write Skew Check.

In the classical literature, the term 'write skew' refers to an anomaly that can arise with **Snapshot Isolation (SI)**.

Snapshot Isolation is a guarantee that a transaction will always read data from a snapshot of the cache store data as of the time the transaction started, which is called its Start-Timestamp.

In the context of Infinispan, you have seen that Infinispan is not implementing Snapshot Isolation, but rather an efficiently weaker consistency level. The key difference with respect to SI is that the grid is storing a single version of each entry, and there is no guarantee that the reads of a given transaction will return from the same snapshot.

Infinispan provides a reliable mechanism of data versioning to improve write skew checks when using optimistic transactions, `REPEATABLE_READ`, and a clustered cache. To enable write skew check, set the `<locking>` element's `writeSkewCheck` attribute to `true` in the config file. The following table describes the attributes of the `<locking>` element:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>concurrencyLevel (ISPN 6)</td>
<td>int</td>
<td>32 (ISPN 6)</td>
<td>You can use this setting to specify a fixed concurrency level for lock containers. It defines the number of available concurrent threads you might have interacting with Infinispan.</td>
</tr>
<tr>
<td>concurrency-level (ISPN 7)</td>
<td>int</td>
<td>1000 (ISPN 7)</td>
<td>You can use this setting to specify a fixed concurrency level for lock containers. It defines the number of available concurrent threads you might have interacting with Infinispan.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Type</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>----------------</td>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>isolationLevel (ISPN 6)</td>
<td>REPEATABLE_READ&lt;br&gt;READ_COMMITTED</td>
<td>READ_COMMITTED</td>
<td>This defines the isolation level for the cache. As we said, Infinispan supports only the REPEATABLE_READ and READ_COMMITTED isolation levels.</td>
</tr>
<tr>
<td>isolation (ISPN 7)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lockAcquisitionTimeout (ISPN 6)</td>
<td>long</td>
<td>10000 (ISPN 6)&lt;br&gt;* 10 seconds&lt;br&gt;15000 (ISPN 7)&lt;br&gt;* 15 seconds</td>
<td>This defines the maximum time (in milliseconds) to attempt a lock acquisition.</td>
</tr>
<tr>
<td>acquire-timeout (ISPN 7)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>useLockStriping (ISPN 6)</td>
<td>Boolean</td>
<td>false</td>
<td>If set to true, Infinispan will maintain a pool of shared locks to be shared by the entries that have to be locked. If set to false, a lock will be created under request, per entry in the cache. Lock striping can help to control the memory footprint of your cache, but may reduce concurrency.</td>
</tr>
<tr>
<td>striping (ISPN 7)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Infinispan provides the LockManager component to deal with all aspects of acquiring and releasing locks for cache entries.

LockManager makes use of the `org.infinispan.util.concurrent.locks.containers.LockContainer` class to acquire a lock for a given key, to get and release a lock. LockContainer has two implementations, one with support for lock striping and support for one lock for each entry.

Lock striping is a technique used in the implementation of ConcurrentHashMap in Java 6 to increase concurrency. The basic idea is that you can break up a cache to different segments (or stripes) and write to these segments concurrently.
Lock striping provides a highly scalable locking mechanism and helps control memory footprint, but you may reduce concurrency in the system and run the risk of blocking irrelevant entries in the same lock.

Lock striping is disabled as a default in Infinispan; to enable lock striping, set the useLockStriping attribute to true in the config file, and you can tune the size of a segment used by lock striping using the concurrencyLevel attribute of the locking configuration element.

For Infinispan 6 the configuration is as follows:

```xml
<namedCache name="transactionCacheWithLocking">
  <jmxStatistics enabled="true"/>
  <transaction transactionManagerLookupClass="org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup" transactionMode="TRANSACTIONAL" lockingMode="PESSIMISTIC"/>
  <locking isolationLevel="READ_COMMITTED" writeSkewCheck="false" concurrencyLevel="5000" useLockStriping="true"/>
</namedCache>
```

The way you configure locking in Infinispan 7 is quite similar to the earlier version:

```xml
<local-cache name="transactionCacheWithLocking">
  <transaction transaction-manager-lookup="org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup" mode="NON_XA" locking="PESSIMISTIC"/>
  <locking isolation="READ_COMMITTED" write-skew="false" concurrency-level="5000" striping="true"/>
</local-cache>
```

If the lock striping attribute is disabled, a lock will be generated based on the hash code of the entry's key, created per entry in the cache, which can increase memory usage, and so on.

Previously, in Infinispan 4.x lock striping was enabled by default. From Infinispan 5.0, due to potential deadlocks, this mechanism is disabled by default.
Implicit and explicit locking

Infinispan provides two lock mechanisms for managing and controlling concurrent access to objects in the grid. These mechanisms can be used as an alternative solution in situations where MVCC does not provide the expected behavior.

Infinispan, by default, uses lazy remote locking, which reduces traffic. Locks are acquired on the local node that is running the transaction while other nodes try during the two-phase commit phase to lock cache entries involved in a transaction. However, since you are working with Infinispan 6, you can configure your cache to (explicitly or implicitly) eagerly lock cache entries.

This feature was removed from the current version of Infinispan (7.*).

The Infinispan cache interface includes basic locking methods, which allows cache users to use these methods during a transaction, to lock the cache entries eagerly.

On lock calls, Infinispan will attempt to lock the requested cache keys across the cluster nodes of the grid and at the commit (or rollback) phase, Infinispan will release all locks held by the transaction, regardless of success or failure.

A cache object can be locked explicitly by the lock method:

```java
    tx.begin()
    cache.lock(K1)    // acquire cluster wide lock on K1
    cache.put(K1,VX)  // guaranteed to succeed
    tx.commit()       // releases locks
```

Implicit locking obtains access rights to cache entries, as they are needed by an application. In general, the implicit locking offered by Infinispan provides a level of concurrency that is sufficient for most applications.

Infinispan will implicitly obtain the appropriate locks for your application at the point at which they are needed, as cache entries are accessed for write operations.

In the following sample transaction, we can see one transaction running in one of the cache nodes:

```java
    tx.begin()
    cache.put(K1,V1)    // acquire cluster wide lock on K1
    cache.put(K2,V2)    // acquire cluster wide lock on K2
```
In a nutshell, for implicit eager locking, Infinispan will check for each modification whether the cache entry is locked locally. If the entry is locked, it means that the entry is also locked in the grid, otherwise, if the entry is not locked Infinispan will send a cluster wide lock request to acquire the lock.

You can also lock a single remote node; however, this configuration is only applied on distributed mode, and would make the number of remote locks acquired to be 1 always, regardless of the configured number of owners.

Infinispan guarantees data consistency in front of a single node lock. The lock for a given key is always deterministically acquired on the same node of the cluster, regardless of where the transaction originates.

**Lock timeouts**

Once you have a lock, you can hold it to execute your required operations, and then, when you finish your tasks you can release the lock for another process to use. You can define the limit of time a cache client can spend waiting to acquire a lock; if a lock request does not return before the specified timeout limit, one of the transactions will rollback, allowing the other to continue working.

You can define a lock acquisition timeout (LAT), to a higher threshold (default is 10 seconds), in the `<locking>` element of your default or named cache configuration. The following example sets the lock acquisition timeout to 20 seconds:

To set the LAT to 20 seconds in Infinispan 6, add the following code:

```xml
<locking lockAcquisitionTimeout="20000"/>
```

To set the LAT to 20 seconds in Infinispan 7, add the following code:

```xml
<locking acquire-timeout="20000"/>
```
Deadlock detection

One risk that might come out with the use of explicit lock is the occurrence of deadlocks, which can occur when concurrent users (two or more) are waiting for an object that has been locked by themselves.

The following situation illustrates a deadlock, imagine we have two transactions and each transaction has a lock on the entry it attempts to update; and the two transactions (in parallel) proceed without committing the transaction. However, each transaction tries to update the cache entry held by the other transaction. As a consequence, both of them will be blocked, because both transactions will not be able to retrieve the entry they need in order to proceed or terminate the transaction.

The following diagram depicts the scenario where two simple transactions, both trying to lock two of the same entries, can get into a deadlock situation:
**Transaction 01** started off by successfully acquiring the lock on entry key K2, with the intent to change it later. Likewise, **Transaction 02** successfully acquired the lock on entry key K1.

Now, in order to continue its processing, **Transaction 01** tries to acquire the lock on K1 as well. But now, **Transaction 02** already locks K1. **Transaction 01** has to wait until **Transaction 02** finishes.

In this scenario, **Transaction 01** patiently waits for **Transaction 02** to finish and to release K1 eventually. But at the same time, **Transaction 02** holds K1, it tries to acquire K2, and cannot get it. This is the most common and simplest scenario of a deadlock, where we have two or more clients (threads) waiting forever because of a locking dependency in all the threads.

User experience may be affected because in all of the following, request(s) to the cache entry will freeze during the deadlock period, which might extend up to LAT.

Neither transaction can obtain the desired key they need in order to proceed or terminate the transaction. The only way out from the deadlock is to break some of the locks by sacrificing at least one transaction, so that another transaction can complete successfully. Also there's a chance that both **Transaction 01** and **Transaction 02** rollback by timing out.

By default, deadlock detection is disabled, but in Infinispan 6, you can enable it for individual caches, under the `namedCache` configuration element, by adding the following:

```xml
<deadlockDetection enabled="true" spinDuration="1000"/>
```

In Infinispan 7, you can enable deadlock detection by specifying the `deadlock-detection-spin` attribute, which defines the time period allowed that an instance can wait to acquire a particular lock:

```xml
<local-cache deadlock-detection-spin="1000"/>
```

An indication that you may need to enable deadlock detection is when you start to see a large number of transactions rolling back and the `TimeoutException` messages. In fact, `TimeoutException` might be caused by other factors too; however, during deadlocks Infinispan will always throw a `TimeoutException`.

Another situation where you should consider using deadlock detection is when you have a high contention on a set of keys, also there are other ways to analyze where deadlock detection is appropriate, but the best method is to monitor and benchmark the server from outside. You can use JMX to monitor and get statistical information such as the number of deadlocks detected, using the `DeadlockDetectingLockManager` MBean. We will see monitoring and management aspects in details in the next chapter.
Data versioning
To configure an efficient write skew check, you can also configure your Infinispan cache to enable versioning and write skew check explicitly using the <versioning> section in your configuration file. Versioning allows concurrency to be managed through MVCC.

The <versioning> element defines only two attributes. The enable attribute determines if versioning is enabled as by default it is disabled, while the versioningScheme attribute defines the versioning scheme Infinispan should use. The possible values are SIMPLE or NONE, the default value is NONE.

When Infinispan is operating in local mode, it's possible to make a more adequate and reliable write skew check using Java object references to compare differences; however, this technique is useless in a distributed environment and a more reliable form of versioning is necessary to provide reliable write skew checks.

The org.infinispan.container.versioning.SimpleClusteredVersion class is an implementation of the proposed org.infinispan.container.versioning.EntryVersion interface, which provides a simple cluster-aware versioning schema, backed by a long version attribute that is incremented each time the cache entry is updated.

By default, versioning is disabled, so pay attention if you are using transactions with write skew checks and REPEATABLE_READ as an isolation level, because it is not reliable if you are using it in a cluster.

To enable versioning in Infinispan 6, add the following code:

```
<versioning enabled="true" versioningScheme="SIMPLE" />
```

To enable versioning in Infinispan 7, add the following code:

```
<versioning scheme="SIMPLE" />
```
Otherwise, if you want, you can create versioning programmatically by adding the following code:

```java
Configuration config = new ConfigurationBuilder()
    .versioning()
    .enable()
    .scheme(VersioningScheme.SIMPLE)
    .transaction()
    .transactionMode(TransactionMode.TRANSACTIONAL)
    .transactionManagerLookup(new GenericTransactionManagerLookup())
    .autoCommit(true)
    .build();
```

**Summary**

In this chapter, we looked at how Infinispan deals with transactions, but first we had an introduction to transaction fundamentals, a glimpse of JTA integration, and how to design your application to use different transactional models, optimistic and pessimistic.

In the second part of the chapter, we had a deeper look on concurrency control mechanisms to ensure data integrity, such as Multi-Version Concurrency Control (MVCC), isolation level, and locking control.

Now that you know how to configure different transaction strategies for your cache, it's time to learn how to monitor problems in production and how to manage your cache instances.
Where to buy this book

You can buy Infinispan Data Grid Platform Definitive Guide from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.