Using the Infinispan Hot Rod Server

Table of Contents

Chapter 1. Using Hot Rod Server

The Infinispan Server distribution contains a server module that implements Infinispan's custom binary protocol called Hot Rod. The protocol was designed to enable faster client/server interactions compared to other existing text based protocols and to allow clients to make more intelligent decisions with regards to load balancing, failover and even data location operations.

To connect to Infinispan over this highly efficient Hot Rod protocol you can either use one of the clients described in this chapter, or use higher level tools such as Hibernate OGM.

1.1. Hot Rod Java clients

Use Java clients to access data through a Infinispan Hot Rod server.

Infinispan provides a reference Hot Rod Java client implementation.

Visit the Hot Rod clients page on *infinispan.org* to download the Java client and find client implementations in other languages.

1.1.1. Programmatically Configuring Hot Rod Java Clients

Use the `ConfigurationBuilder` class to generate immutable configuration objects that you can pass to `RemoteCacheManager`.

For example, create a client instance with the Java fluent API as follows:

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb
      = new org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.marshaller(new org.infinispan.commons.marshall.ProtoStreamMarshaller())
  .statistics()
      .enable()
      .jmxDomain("org.infinispan")
  .addServer()
      .host("127.0.0.1")
      .port(11222);
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```

Reference

org.infinispan.client.hotrod.configuration.ConfigurationBuilder

1.1.2. Configuring Hot Rod Java Client Property Files

Add `hotrod-client.properties` to your classpath so that the client passes configuration to `RemoteCacheManager`.

To use `hotrod-client.properties` somewhere other than your classpath, do:

```
ConfigurationBuilder b = new ConfigurationBuilder();
Properties p = new Properties();
try(Reader r = new FileReader("/path/to/hotrod-client.properties")) {
   p.load(r);
   b.withProperties(p);
}
RemoteCacheManager rcm = new RemoteCacheManager(b.build());
```

Example hotrod-client.properties

```
# Hot Rod client configuration
infinispan.client.hotrod.server_list = 127.0.0.1:11222
infinispan.client.hotrod.marshaller =
org.infinispan.commons.marshall.ProtoStreamMarshaller
infinispan.client.hotrod.async_executor_factory =
org.infinispan.client.hotrod.impl.async.DefaultAsyncExecutorFactory
infinispan.client.hotrod.default_executor_factory.pool_size = 1
infinispan.client.hotrod.hash_function_impl.2 =
org.infinispan.client.hotrod.impl.consistenthash.ConsistentHashV2
infinispan.client.hotrod.tcp_no_delay = true
infinispan.client.hotrod.tcp_keep_alive = false
infinispan.client.hotrod.request_balancing_strategy =
org.infinispan.client.hotrod.impl.transport.tcp.RoundRobinBalancingStrategy
infinispan.client.hotrod.key_size_estimate = 64
infinispan.client.hotrod.value_size_estimate = 512
infinispan.client.hotrod.force_return_values = false

## Connection pooling configuration
maxActive = -1
maxIdle = -1
whenExhaustedAction = 1
minEvictableIdleTimeMillis=300000
minIdle = 1
```

Substitute Java system properties to replace values at runtime, for example:

```
infinispan.client.hotrod.server_list = ${server_list}
```

The preceding example expands the value of the `infinispan.client.hotrod.server_list` property to the value of the `server_list` Java system property, which comes from system property named `jboss.bind.address.management` or defaults to `127.0.0.1`.

Reference

- org.infinispan.client.hotrod.configuration lists and describes Hot Rod client properties.
- org.infinispan.client.hotrod.RemoteCacheManager

- Java system properties

1.1.3. Authentication Mechanisms

Infinispan servers can require authentication for client connections with different mechanisms.

SCRAM

Provides username and password authentication with the (*"default"*) realm on Infinispan servers.

SCRAM is recommended approach for simple authentication.

SCRAM-SHA-512 Hot Rod client configuration

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .authentication()
         .username("myuser")
         .password("qwer1234!");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

DIGEST

Deprecated by SCRAM and available only for backwards compatibility with previous versions of Infinispan server that do not support SCRAM.

DIGEST-MD5 Hot Rod client configuration

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .authentication()
         .saslMechanism("DIGEST-MD5")
         .username("myuser")
         .password("qwer1234!");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

PLAIN

Sends credentials in clear-text to authenticate with Infinispan servers.

PLAIN is recommended only for secure connections that use TLS encryption.

PLAIN Hot Rod client configuration

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .authentication()
         .saslMechanism("PLAIN")
         .username("myuser")
         .password("qwer1234!");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

OAUTHBEARER

Requires client applications to retrieve authentication tokens from OAuth2 providers.

OAUTHBEARER Hot Rod client configuration

```
String token = "..."; // Obtain the token from your OAuth2 provider
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .authentication()
         .saslMechanism("OAUTHBEARER")
         .token(token);
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

You should configure clients with a `TokenCallbackHandler` to refresh OAuth2 tokens before they expire.

OAUTHBEARER authentication with TokenCallbackHandler

```
String token = "..."; // Obtain the token from your OAuth2 provider
TokenCallbackHandler tokenHandler = new TokenCallbackHandler(token);
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .authentication()
         .saslMechanism("OAUTHBEARER")
         .callbackHandler(tokenHandler);
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
// Refresh the token
tokenHandler.setToken("newToken");
```

EXTERNAL

Uses certificates to authenticate with Infinispan servers. You provide clients with truststores to validate server certificates and keystores to supply client certificates.

EXTERNAL Hot Rod client configuration

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .ssl()
         // TrustStore stores trusted CA certificates for the server.
         .trustStoreFileName("/path/to/truststore")
         .trustStorePassword("truststorepassword".toCharArray())
         // KeyStore stores valid client certificates.
         .keyStoreFileName("/path/to/keystore")
         .keyStorePassword("keystorepassword".toCharArray())
      .authentication()
         .saslMechanism("EXTERNAL");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

Reference

Setting Up Encryption

GSSAPI and Kerberos

Uses the Kerberos protocol to provide clients with authentication tokens for centralized security

services.

To use the GSSAPI mechanism, you must create a *LoginContext* for clients to obtain a Ticket Granting Ticket (TGT) as follows:

1. Define a login module in a login configuration file.

   gss.conf

   ```
   GssExample {
       com.sun.security.auth.module.Krb5LoginModule required client=TRUE;
   };
   ```

   For the IBM JDK:

   gss-ibm.conf

   ```
   GssExample {
       com.ibm.security.auth.module.Krb5LoginModule required client=TRUE;
   };
   ```

2. Set the following system properties:

   ```
   java.security.auth.login.config=gss.conf

   java.security.krb5.conf=/etc/krb5.conf
   ```

   `krb5.conf` provides the location of your KDC. Use the *kinit* command to authenticate with Kerberos and verify `krb5.conf`.

3. Configure your Hot Rod client for GSSAPI.

GSSAPI Hot Rod client configuration

```
LoginContext lc = new LoginContext("GssExample", new BasicCallbackHandler("krb_user",
"krb_password".toCharArray()));
lc.login();
Subject clientSubject = lc.getSubject();

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .authentication()
         .enable()
         .saslMechanism("GSSAPI")
         .clientSubject(clientSubject)
         .callbackHandler(new BasicCallbackHandler());
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

The preceding configuration uses the `BasicCallbackHandler` to retrieve the client subject and handle authentication. However, this actually invokes different callbacks.

`NameCallback` and `PasswordCallback` construct the client subject.

`AuthorizeCallback` is called during SASL authentication.

Custom CallbackHandler

Hot Rod clients set up a default `CallbackHandler` to pass credentials to SASL mechanisms. In some cases, you might need to provide a custom `CallbackHandler`.

Your `CallbackHandler` needs to handle callbacks that are specific to the authentication mechanism that you use. However, it is beyond the scope of this document to provide examples for each possible callback type.

Custom CallbackHandler in Hot Rod client configuration

```
public class MyCallbackHandler implements CallbackHandler {
   final private String username;
   final private char[] password;
   final private String realm;

   public MyCallbackHandler(String username, String realm, char[] password) {
      this.username = username;
      this.password = password;
      this.realm = realm;
```

```
   }

   @Override
   public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
      for (Callback callback : callbacks) {
         if (callback instanceof NameCallback) {
            NameCallback nameCallback = (NameCallback) callback;
            nameCallback.setName(username);
         } else if (callback instanceof PasswordCallback) {
            PasswordCallback passwordCallback = (PasswordCallback) callback;
            passwordCallback.setPassword(password);
         } else if (callback instanceof AuthorizeCallback) {
            AuthorizeCallback authorizeCallback = (AuthorizeCallback) callback;
            authorizeCallback.setAuthorized(authorizeCallback.getAuthenticationID()
.equals(
                  authorizeCallback.getAuthorizationID()));
         } else if (callback instanceof RealmCallback) {
            RealmCallback realmCallback = (RealmCallback) callback;
            realmCallback.setText(realm);
         } else {
            throw new UnsupportedCallbackException(callback);
         }
      }
   }
}

   ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .authentication()
         .enable()
         .serverName("myhotrodserver")
         .saslMechanism("DIGEST-MD5")
         .callbackHandler(new MyCallbackHandler("myuser","default","qwer1234!"
.toCharArray()));
remoteCacheManager=new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache=remoteCacheManager.getCache("secured");
```

1.1.4. Setting Up Encryption

Encryption uses TLS/SSL, so it requires setting up an appropriate server certificate chain. Generally, a certificate chain looks like the following:

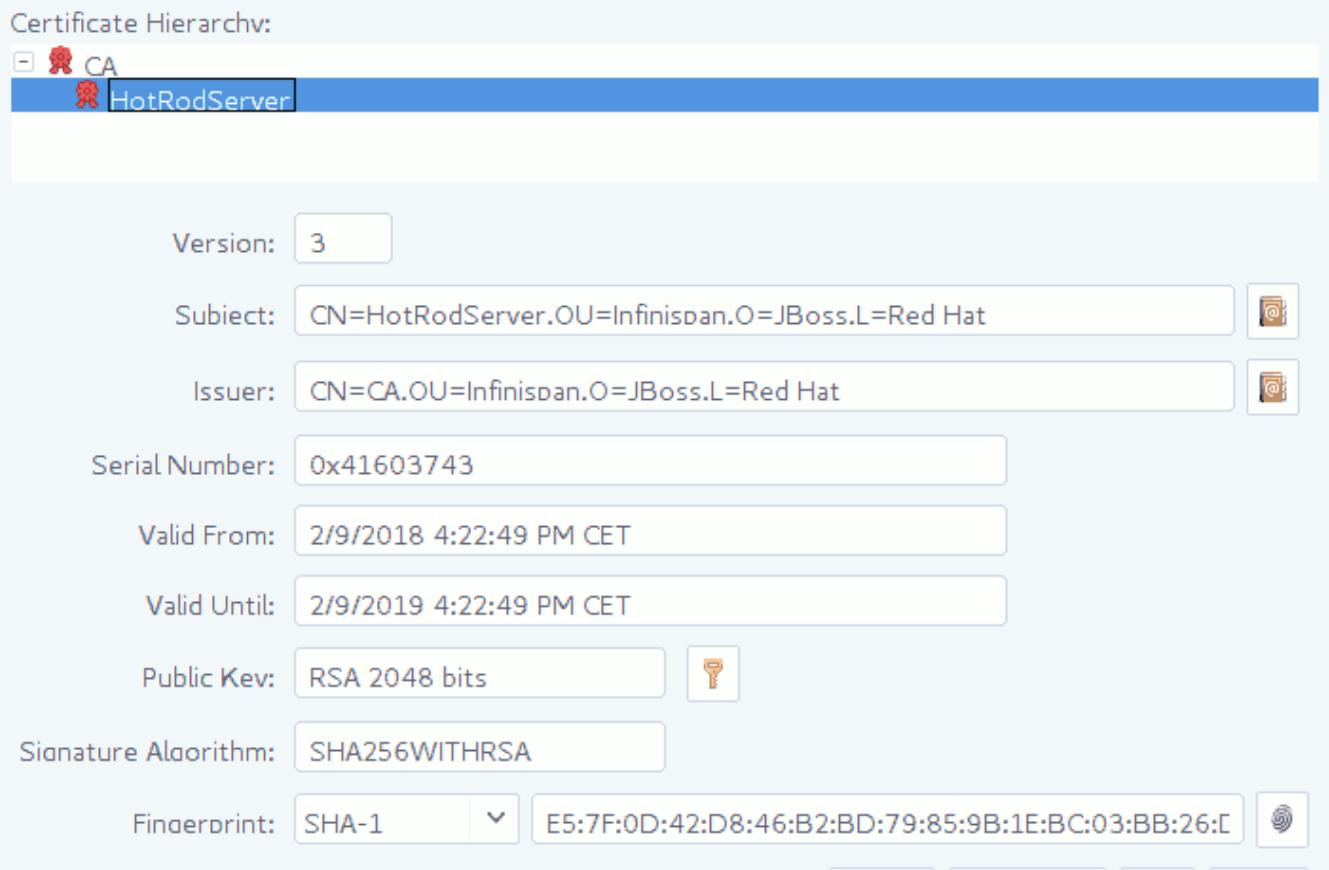


Figure 1. Certificate chain

In the above example there is one certificate authority *"CA"* which has issued a certificate for *"HotRodServer"*. In order for a client to trust the server, it needs at least a portion of the above chain (usually, just the public certificate for *"CA"*). This certificate needs to placed in a keystore and used as a *TrustStore* on the client and used as shown below:

Hot Rod client configuration with TLS (server cert)

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .ssl()
         // TrustStore is a KeyStore which contains part of the server certificate
chain (e.g. the CA Root public cert)
         .trustStoreFileName("/path/to/truststore")
         .trustStorePassword("truststorepassword".toCharArray());
RemoteCache<String, String> cache=remoteCacheManager.getCache("secured");
```

SNI

The server may have been configured with TLS/SNI support (Server Name Indication). This means that the server is presenting multiple identities (probably bound to separate cache containers). The client can specify which identity to connect to by specifying its name:

Hot Rod client configuration with SNI (server cert)

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .ssl()
         .sniHostName("myservername")
         // TrustStore is a KeyStore which contains part of the server certificate
chain (e.g. the CA Root public cert)
         .trustStoreFileName("/path/to/truststore")
         .trustStorePassword("truststorepassword".toCharArray());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

Client certificates

With the above configurations the client trusts the server. For increased security, a server administrator may have set up the server to require the client to offer a valid certificate for mutual trust. This kind of configuration requires the client to present its own certificate, usually issued by the same certificate authority as the server. This certificate must be stored in a keystore and used as follows:

Hot Rod client configuration with TLS (server and client cert)

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
   .addServer()
      .host("127.0.0.1")
      .port(11222)
   .security()
      .ssl()
         // TrustStore is a KeyStore which contains part of the server certificate
chain (e.g. the CA Root public cert)
         .trustStoreFileName("/path/to/truststore")
         .trustStorePassword("truststorepassword".toCharArray())
         // KeyStore containing this client's own certificate
         .keyStoreFileName("/path/to/keystore")
         .keyStorePassword("keystorepassword".toCharArray())
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

Please read the KeyTool documentation for more details on KeyStores. Additionally, the KeyStore Explorer is a great GUI tool for easily managing KeyStores.

1.1.5. Basic API

Below is a sample code snippet on how the client API can be used to store or retrieve information from a Hot Rod server using the Java Hot Rod client. It assumes that a Hot Rod server has been

started bound to the default location (localhost:11222)

```
//API entry point, by default it connects to localhost:11222
CacheContainer cacheContainer = new RemoteCacheManager();

//obtain a handle to the remote default cache
Cache<String, String> cache = cacheContainer.getCache();

//now add something to the cache and make sure it is there
cache.put("car", "ferrari");
assert cache.get("car").equals("ferrari");

//remove the data
cache.remove("car");
assert !cache.containsKey("car") : "Value must have been removed!";
```

The client API maps the local API: RemoteCacheManager corresponds to DefaultCacheManager (both implement CacheContainer). This common API facilitates an easy migration from local calls to remote calls through Hot Rod: all one needs to do is switch between DefaultCacheManager and RemoteCacheManager - which is further simplified by the common CacheContainer interface that both inherit.

1.1.6. RemoteCache(.keySet|.entrySet|.values)

The collection methods `keySet`, `entrySet` and `values` are backed by the remote cache. That is that every method is called back into the `RemoteCache`. This is useful as it allows for the various keys, entries or values to be retrieved lazily, and not requiring them all be stored in the client memory at once if the user does not want. These collections adhere to the `Map` specification being that `add` and `addAll` are not supported but all other methods are supported.

One thing to note is the `Iterator.remove` and `Set.remove` or `Collection.remove` methods require more than 1 round trip to the server to operate. You can check out the RemoteCache Javadoc to see more details about these and the other methods.

Iterator Usage

The iterator method of these collections uses `retrieveEntries` internally, which is described below. If you notice `retrieveEntries` takes an argument for the batch size. There is no way to provide this to the iterator. As such the batch size can be configured via system property `infinispan.client.hotrod.batch_size` or through the ConfigurationBuilder when configuring the `RemoteCacheManager`.

Also the `retrieveEntries` iterator returned is `Closeable` as such the iterators from `keySet`, `entrySet` and `values` return an `AutoCloseable` variant. Therefore you should always close these `Iterator`s when you are done with them.

```
try (CloseableIterator<Entry<K, V>> iterator = remoteCache.entrySet().iterator) {
 ...
}
```

What if I want a deep copy and not a backing collection?

Previous version of `RemoteCache` allowed for the retrieval of a deep copy of the `keySet`. This is still possible with the new backing map, you just have to copy the contents yourself. Also you can do this with `entrySet` and `values`, which we didn't support before.

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

Please use extreme cautiong with this as a large number of keys can and will cause OutOfMemoryError in the client.

```
Set keys = remoteCache.keySet();
```

1.1.7. Remote Iterator

Alternatively, if memory is a concern (different batch size) or you wish to do server side filtering or conversion), use the remote iterator api to retrieve entries from the server. With this method you can limit the entries that are retrieved or even returned a converted value if you dont' need all properties of your entry.

```
// Retrieve all entries in batches of 1000
int batchSize = 1000;
try (CloseableIterator<Entry<Object, Object>> iterator = remoteCache.retrieveEntries
(null, batchSize)) {
     while(iterator.hasNext()) {
        // Do something
     }
}

// Filter by segment
Set<Integer> segments = ...
try (CloseableIterator<Entry<Object, Object>> iterator = remoteCache.retrieveEntries
(null, segments, batchSize)) {
     while(iterator.hasNext()) {
        // Do something
     }
}

// Filter by custom filter
try (CloseableIterator<Entry<Object, Object>> iterator = remoteCache.retrieveEntries(
"myFilterConverterFactory", segments, batchSize)) {
     while(iterator.hasNext()) {
        // Do something
     }
}
```

In order to use custom filters, it's necessary to deploy them first in the server. Follow the steps:

- Create a factory for the filter extending KeyValueFilterConverterFactory, annotated with @NamedFactory containing the name of the factory, example:

```
import java.io.Serializable;

import org.infinispan.filter.AbstractKeyValueFilterConverter;
import org.infinispan.filter.KeyValueFilterConverter;
import org.infinispan.filter.KeyValueFilterConverterFactory;
import org.infinispan.filter.NamedFactory;
import org.infinispan.metadata.Metadata;

@NamedFactory(name = "myFilterConverterFactory")
public class MyKeyValueFilterConverterFactory implements
KeyValueFilterConverterFactory {

   @Override
   public KeyValueFilterConverter<String, SampleEntity1, SampleEntity2>
getFilterConverter() {
      return new MyKeyValueFilterConverter();
   }
   // Filter implementation. Should be serializable or externalizable for DIST caches
   static class MyKeyValueFilterConverter extends AbstractKeyValueFilterConverter
<String, SampleEntity1, SampleEntity2> implements Serializable {
      @Override
      public SampleEntity2 filterAndConvert(String key, SampleEntity1 entity, Metadata
metadata) {
         // returning null will case the entry to be filtered out
         // return SampleEntity2 will convert from the cache type SampleEntity1
      }

      @Override
      public MediaType format() {
         // returns the MediaType that data should be presented to this converter.
         // When ommitted, the server will use "application/x-java-object".
         // Returning null will cause the filter/converter to be done in the storage
format.
      }
   }
}
```

- Create a jar with a `META-INF/services/org.infinispan.filter.KeyValueFilterConverterFactory` file and within it, write the fully qualified class name of the filter factory class implementation.
- Optional: If the filter uses custom key/value classes, these must be included in the JAR so that the filter can correctly unmarshall key and/or value instances.
- Deploy the JAR file in the Infinispan Server.

1.1.8. Versioned API

A RemoteCacheManager provides instances of RemoteCache interface that represents a handle to the named or default cache on the remote cluster. API wise, it extends the Cache interface to which it also adds some new methods, including the so called versioned API. Please find below some

examples of this API link:#server_hotrod_failover[but to understand the motivation behind it, make sure you read this section.

The code snippet bellow depicts the usage of these versioned methods:

```
// To use the versioned API, remote classes are specifically needed
RemoteCacheManager remoteCacheManager = new RemoteCacheManager();
RemoteCache<String, String> cache = remoteCacheManager.getCache();

remoteCache.put("car", "ferrari");
RemoteCache.VersionedValue valueBinary = remoteCache.getVersioned("car");

// removal only takes place only if the version has not been changed
// in between. (a new version is associated with 'car' key on each change)
assert remoteCache.remove("car", valueBinary.getVersion());
assert !cache.containsKey("car");
```

In a similar way, for replace:

```
remoteCache.put("car", "ferrari");
RemoteCache.VersionedValue valueBinary = remoteCache.getVersioned("car");
assert remoteCache.replace("car", "lamborghini", valueBinary.getVersion());
```

For more details on versioned operations refer to RemoteCache 's javadoc.

1.1.9. Streaming API

When sending / receiving large objects, it might make sense to stream them between the client and the server. The Streaming API implements methods similar to the Hot Rod Basic API and Hot Rod Versioned API described above but, instead of taking the value as a parameter, they return instances of InputStream and OutputStream. The following example shows how one would write a potentially large object:

```
RemoteStreamingCache<String> streamingCache = remoteCache.streaming();
OutputStream os = streamingCache.put("a_large_object");
os.write(...);
os.close();
```

Reading such an object through streaming:

```
RemoteStreamingCache<String> streamingCache = remoteCache.streaming();
InputStream is = streamingCache.get("a_large_object");
for(int b = is.read(); b >= 0; b = is.read()) {
   ...
}
is.close();
```

The streaming API does **not** apply marshalling/unmarshalling to the values. For this reason you cannot access the same entries using both the streaming and non-streaming API at the same time, unless you provide your own marshaller to detect this situation.

The InputStream returned by the `RemoteStreamingCache.get(K key)` method implements the `VersionedMetadata` interface, so you can retrieve version and expiration information:

```
RemoteStreamingCache<String> streamingCache = remoteCache.streaming();
InputStream is = streamingCache.get("a_large_object");
int version = ((VersionedMetadata) is).getVersion();
for(int b = is.read(); b >= 0; b = is.read()) {
   ...
}
is.close();
```

Conditional write methods (`putIfAbsent`, `replace`) only perform the actual condition check once the value has been completely sent to the server (i.e. when the `close()` method has been invoked on the `OutputStream`.

1.1.10. Creating Event Listeners

Java Hot Rod clients can register listeners to receive cache-entry level events. Cache entry created, modified and removed events are supported.

Creating a client listener is very similar to embedded listeners, except that different annotations and event classes are used. Here's an example of a client listener that prints out each event received:

```
import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener
public class EventPrintListener {

   @ClientCacheEntryCreated
   public void handleCreatedEvent(ClientCacheEntryCreatedEvent e) {
      System.out.println(e);
   }

   @ClientCacheEntryModified
   public void handleModifiedEvent(ClientCacheEntryModifiedEvent e) {
      System.out.println(e);
   }

   @ClientCacheEntryRemoved
   public void handleRemovedEvent(ClientCacheEntryRemovedEvent e) {
      System.out.println(e);
   }

}
```

`ClientCacheEntryCreatedEvent` and `ClientCacheEntryModifiedEvent` instances provide information on the affected key, and the version of the entry. This version can be used to invoke conditional operations on the server, such as `replaceWithVersion` or `removeWithVersion`.

`ClientCacheEntryRemovedEvent` events are only sent when the remove operation succeeds. In other words, if a remove operation is invoked but no entry is found or no entry should be removed, no event is generated. Users interested in removed events, even when no entry was removed, can develop event customization logic to generate such events. More information can be found in the customizing client events section.

All `ClientCacheEntryCreatedEvent`, `ClientCacheEntryModifiedEvent` and `ClientCacheEntryRemovedEvent` event instances also provide a `boolean isCommandRetried()` method that will return true if the write command that caused this had to be retried again due to a topology change. This could be a sign that this event has been duplicated or another event was dropped and replaced (eg: ClientCacheEntryModifiedEvent replaced ClientCacheEntryCreatedEvent).

Once the client listener implementation has been created, it needs to be registered with the server. To do so, execute:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener());
```

1.1.11. Removing Event Listeners

When an client event listener is not needed any more, it can be removed:

```
EventPrintListener listener = ...
cache.removeClientListener(listener);
```

1.1.12. Filtering Events

In order to avoid inundating clients with events, users can provide filtering functionality to limit the number of events fired by the server for a particular client listener. To enable filtering, a cache event filter factory needs to be created that produces filter instances:

```
import org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory;
import org.infinispan.filter.NamedFactory;

@NamedFactory(name = "static-filter")
class StaticCacheEventFilterFactory implements CacheEventFilterFactory {
   @Override
   public CacheEventFilterFactory<Integer, String> getFilter(Object[] params) {
      return new StaticCacheEventFilter();
   }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class StaticCacheEventFilter implements CacheEventFilter<Integer, String>,
Serializable {
   @Override
   public boolean accept(Integer key, String oldValue, Metadata oldMetadata,
         String newValue, Metadata newMetadata, EventType eventType) {
      if (key.equals(1)) // static key
         return true;

      return false;
   }
}
```

The cache event filter factory instance defined above creates filter instances which statically filter out all entries except the one whose key is 1.

To be able to register a listener with this cache event filter factory, the factory has to be given a unique name, and the Hot Rod server needs to be plugged with the name and the cache event filter factory instance. Plugging the Infinispan Server with a custom filter involves the following steps:

1. Create a JAR file with the filter implementation within it.
2. Optional: If the cache uses custom key/value classes, these must be included in the JAR so that the callbacks can be executed with the correctly unmarshalled key and/or value instances. If the

client listener has `useRawData` enabled, this is not necessary since the callback key/value instances will be provided in binary format.

3. Create a `META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory` file within the JAR file and within it, write the fully qualified class name of the filter class implementation.
4. Deploy the JAR file in the Infinispan Server.

On top of that, the client listener needs to be linked with this cache event filter factory by adding the factory's name to the @ClientListener annotation:

```
@ClientListener(filterFactoryName = "static-filter")
public class EventPrintListener { ... }
```

And, register the listener with the server:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener());
```

Dynamic filter instances that filter based on parameters provided when the listener is registered are also possible. Filters use the parameters received by the filter factories to enable this option. For example:

```
import org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventFilter;

class DynamicCacheEventFilterFactory implements CacheEventFilterFactory {
   @Override
   public CacheEventFilter<Integer, String> getFilter(Object[] params) {
      return new DynamicCacheEventFilter(params);
   }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class DynamicCacheEventFilter implements CacheEventFilter<Integer, String>,
Serializable {
   final Object[] params;

   DynamicCacheEventFilter(Object[] params) {
      this.params = params;
   }

   @Override
   public boolean accept(Integer key, String oldValue, Metadata oldMetadata,
         String newValue, Metadata newMetadata, EventType eventType) {
      if (key.equals(params[0])) // dynamic key
         return true;

      return false;
   }
}
```

The dynamic parameters required to do the filtering are provided when the listener is registered:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener(), new Object[]{1}, null);
```

Filter instances have to marshallable when they are deployed in a cluster so that the filtering can happen right where the event is generated, even if the even is generated in a different node to where the listener is registered. To make them marshallable, either make them extend `Serializable`, `Externalizable`, or provide a custom `Externalizer` for them.

Skipping Notifications

Include the `SKIP_LISTENER_NOTIFICATION` flag when calling remote API methods to perform operations without getting event notifications from the server. For example, to prevent listener notifications when creating or modifying values, set the flag as follows:

```
remoteCache.withFlags(Flag.SKIP_LISTENER_NOTIFICATION).put(1, "one");
```

1.1.13. Customizing Events

The events generated by default contain just enough information to make the event relevant but they avoid cramming too much information in order to reduce the cost of sending them. Optionally, the information shipped in the events can be customised in order to contain more information, such as values, or to contain even less information. This customization is done with `CacheEventConverter` instances generated by a `CacheEventConverterFactory`:

```
import org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventConverter;
import org.infinispan.filter.NamedFactory;

@NamedFactory(name = "static-converter")
class StaticConverterFactory implements CacheEventConverterFactory {
   final CacheEventConverter<Integer, String, CustomEvent> staticConverter = new
StaticCacheEventConverter();
   public CacheEventConverter<Integer, String, CustomEvent> getConverter(final
Object[] params) {
      return staticConverter;
   }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class StaticCacheEventConverter implements CacheEventConverter<Integer, String,
CustomEvent>, Serializable {
   public CustomEvent convert(Integer key, String oldValue, Metadata oldMetadata,
String newValue, Metadata newMetadata, EventType eventType) {
      return new CustomEvent(key, newValue);
   }
}

// Needs to be Serializable, Externalizable or marshallable with Infinispan
Externalizers
// regardless of cluster or local caches
static class CustomEvent implements Serializable {
   final Integer key;
   final String value;
   CustomEvent(Integer key, String value) {
      this.key = key;
      this.value = value;
   }
}
```

In the example above, the converter generates a new custom event which includes the value as well as the key in the event. This will result in bigger event payloads compared with default events, but

if combined with filtering, it can reduce its network bandwidth cost.

The target type of the converter must be either `Serializable` or `Externalizable`. In this particular case of converters, providing an Externalizer will not work by default since the default Hot Rod client marshaller does not support them.

Handling custom events requires a slightly different client listener implementation to the one demonstrated previously. To be more precise, it needs to handle `ClientCacheEntryCustomEvent` instances:

```
import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener
public class CustomEventPrintListener {

   @ClientCacheEntryCreated
   @ClientCacheEntryModified
   @ClientCacheEntryRemoved
   public void handleCustomEvent(ClientCacheEntryCustomEvent<CustomEvent> e) {
      System.out.println(e);
   }

}
```

The `ClientCacheEntryCustomEvent` received in the callback exposes the custom event via `getEventData` method, and the `getType` method provides information on whether the event generated was as a result of cache entry creation, modification or removal.

Similar to filtering, to be able to register a listener with this converter factory, the factory has to be given a unique name, and the Hot Rod server needs to be plugged with the name and the cache event converter factory instance. Plugging the Infinispan Server with an event converter involves the following steps:

1. Create a JAR file with the converter implementation within it.
2. Optional: If the cache uses custom key/value classes, these must be included in the JAR so that the callbacks can be executed with the correctly unmarshalled key and/or value instances. If the client listener has `useRawData` enabled, this is not necessary since the callback key/value instances will be provided in binary format.
3. Create a `META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory` file within the JAR file and within it, write the fully qualified class name of the converter class implementation.
4. Deploy the JAR file in the Infinispan Server.

On top of that, the client listener needs to be linked with this converter factory by adding the factory's name to the @ClientListener annotation:

```
@ClientListener(converterFactoryName = "static-converter")
public class CustomEventPrintListener { ... }
```

And, register the listener with the server:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new CustomEventPrintListener());
```

Dynamic converter instances that convert based on parameters provided when the listener is registered are also possible. Converters use the parameters received by the converter factories to enable this option. For example:

```
import org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventConverter;

@NamedFactory(name = "dynamic-converter")
class DynamicCacheEventConverterFactory implements CacheEventConverterFactory {
   public CacheEventConverter<Integer, String, CustomEvent> getConverter(final
Object[] params) {
      return new DynamicCacheEventConverter(params);
   }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers needed
when running in a cluster
class DynamicCacheEventConverter implements CacheEventConverter<Integer, String,
CustomEvent>, Serializable {
   final Object[] params;

   DynamicCacheEventConverter(Object[] params) {
      this.params = params;
   }

   public CustomEvent convert(Integer key, String oldValue, Metadata oldMetadata,
         String newValue, Metadata newMetadata, EventType eventType) {
      // If the key matches a key given via parameter, only send the key information
      if (params[0].equals(key))
         return new CustomEvent(key, null);

      return new CustomEvent(key, newValue);
   }
}
```

The dynamic parameters required to do the conversion are provided when the listener is registered:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener(), null, new Object[]{1});
```

Converter instances have to marshallable when they are deployed in a cluster, so that the conversion can happen right where the event is generated, even if the even is generated in a different node to where the listener is registered. To make them marshallable, either make them extend `Serializable`, `Externalizable`, or provide a custom `Externalizer` for them.

1.1.14. Filter and Custom Events

If you want to do both event filtering and customization, it's easier to implement `org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverter` which allows both filter and customization to happen in a single step. For convenience, it's recommended to extend `org.infinispan.notifications.cachelistener.filter.AbstractCacheEventFilterConverter` instead of implementing `org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverter` directly. For example:

```
import org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventConverter;

@NamedFactory(name = "dynamic-filter-converter")
class DynamicCacheEventFilterConverterFactory implements
CacheEventFilterConverterFactory {
   public CacheEventFilterConverter<Integer, String, CustomEvent> getFilterConverter
(final Object[] params) {
      return new DynamicCacheEventFilterConverter(params);
   }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers needed
when running in a cluster
//
class DynamicCacheEventFilterConverter extends AbstractCacheEventFilterConverter
<Integer, String, CustomEvent>, Serializable {
   final Object[] params;

   DynamicCacheEventFilterConverter(Object[] params) {
      this.params = params;
   }

   public CustomEvent filterAndConvert(Integer key, String oldValue, Metadata
oldMetadata,
         String newValue, Metadata newMetadata, EventType eventType) {
      // If the key matches a key given via parameter, only send the key information
      if (params[0].equals(key))
         return new CustomEvent(key, null);

      return new CustomEvent(key, newValue);
   }
}
```

Similar to filters and converters, to be able to register a listener with this combined filter/converter factory, the factory has to be given a unique name via the `@NamedFactory` annotation, and the Hot Rod server needs to be plugged with the name and the cache event converter factory instance. Plugging the Infinispan Server with an event converter involves the following steps:

1. Create a JAR file with the converter implementation within it.
2. Optional: If the cache uses custom key/value classes, these must be included in the JAR so that the callbacks can be executed with the correctly unmarshalled key and/or value instances. If the client listener has `useRawData` enabled, this is not necessary since the callback key/value instances will be provided in binary format.
3. Create a `META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverterFactory` file within the JAR file and within it, write the fully qualified class name of the converter class implementation.

4. Deploy the JAR file in the Infinispan Server.

From a client perspective, to be able to use the combined filter and converter class, the client listener must define the same filter factory and converter factory names, e.g.:

```
@ClientListener(filterFactoryName = "dynamic-filter-converter", converterFactoryName =
"dynamic-filter-converter")
public class CustomEventPrintListener { ... }
```

The dynamic parameters required in the example above are provided when the listener is registered via either filter or converter parameters. If filter parameters are non-empty, those are used, otherwise, the converter parameters:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new CustomEventPrintListener(), new Object[]{1}, null);
```

1.1.15. Event Marshalling

Hot Rod servers can store data in different formats, but in spite of that, Java Hot Rod client users can still develop `CacheEventConverter` or `CacheEventFilter` instances that work on typed objects. By default, filters and converter will use data as POJO (application/x-java-object) but it is possible to override the desired format by overriding the method `format()` from the filter/converter. If the format returns `null`, the filter/converter will receive data as it's stored.

As indicated in the Marshalling Data section, Hot Rod Java clients can be configured to use a different `org.infinispan.commons.marshall.Marshaller` instance. If doing this and deploying `CacheEventConverter` or `CacheEventFilter` instances, to be able to present filters/converter with Java Objects rather than marshalled content, the server needs to be able to convert between objects and the binary format produced by the marshaller.

To deploy a Marshaller instance server-side, follow a similar method to the one used to deploy `CacheEventConverter` or `CacheEventFilter` instances:

1. Create a JAR file with the converter implementation within it.
2. Create a `META-INF/services/org.infinispan.commons.marshall.Marshaller` file within the JAR file and within it, write the fully qualified class name of the marshaller class implementation.
3. Deploy the JAR file in the Infinispan Server.

Note that the Marshaller could be deployed in either a separate jar, or in the same jar as the `CacheEventConverter` and/or `CacheEventFilter` instances.

Deploying Protostream Marshallers

If a cache stores Protobuf content, as it happens when using ProtoStream marshaller in the Hot Rod client, it's not necessary to deploy a custom marshaller since the format is already support by the server: there are transcoders from Protobuf format to most common formats like JSON and POJO.

When using filters/converters with those caches, and it's desirable to use filter/converters with Java Objects rather binary Protobuf data, it's necessary to configure the extra ProtoStream marshallers so that the server can unmarshall the data before filtering/converting. To do so, you must configure the required SerializationContextInitializer(s) as part of the Infinispan server configuration.

See ProtoStream for more information.

1.1.16. Listener State Handling

Client listener annotation has an optional `includeCurrentState` attribute that specifies whether state will be sent to the client when the listener is added or when there's a failover of the listener.

By default, `includeCurrentState` is false, but if set to true and a client listener is added in a cache already containing data, the server iterates over the cache contents and sends an event for each entry to the client as a `ClientCacheEntryCreated` (or custom event if configured). This allows clients to build some local data structures based on the existing content. Once the content has been iterated over, events are received as normal, as cache updates are received. If the cache is clustered, the entire cluster wide contents are iterated over.

`includeCurrentState` also controls whether state is received when the node where the client event listener is registered fails and it's moved to a different node. The next section discusses this topic in depth.

1.1.17. Listener Failure Handling

When a Hot Rod client registers a client listener, it does so in a single node in a cluster. If that node fails, the Java Hot Rod client detects that transparently and fails over all listeners registered in the node that failed to another node.

During this fail over the client might miss some events. To avoid missing these events, the client listener annotation contains an optional parameter called `includeCurrentState` which if set to true, when the failover happens, the cache contents can iterated over and `ClientCacheEntryCreated` events (or custom events if configured) are generated. By default, `includeCurrentState` is set to false.

Java Hot Rod clients can be made aware of such fail over event by adding a callback to handle it:

```
@ClientCacheFailover
public void handleFailover(ClientCacheFailoverEvent e) {
  ...
}
```

This is very useful in use cases where the client has cached some data, and as a result of the fail over, taking in account that some events could be missed, it could decide to clear any locally cached data when the fail over event is received, with the knowledge that after the fail over event, it will receive events for the contents of the entire cache.

1.1.18. Near Caching

The Java Hot Rod client can be optionally configured with a near cache, which means that the Hot Rod client can keep a local cache that stores recently used data. Enabling near caching can significantly improve the performance of read operations `get` and `getVersioned` since data can potentially be located locally within the Hot Rod client instead of having to go remote.

To enable near caching, the user must set the near cache mode to `INVALIDATED`. By doing that near cache is populated upon retrievals from the server via calls to `get` or `getVersioned` operations. When near cached entries are updated or removed server-side, the cached near cache entries are invalidated. If a key is requested after it's been invalidated, it'll have to be re-fetched from the server.

You should not use `maxIdle` expiration with near caches, as near-cache reads will not propagate the last access change to the server and to the other clients.

When near cache is enabled, its size must be configured by defining the maximum number of entries to keep in the near cache. When the maximum is reached, near-cached entries are evicted. If providing `0` or a negative value, it is assumed that the near cache is unbounded.

Users should be careful when configuring near cache to be unbounded since it shifts the responsibility to keep the near cache's size within the boundaries of the client JVM to the user.

The Hot Rod client's near cache mode is configured using the `NearCacheMode` enumeration and calling:

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.configuration.NearCacheMode;
...

// Unbounded invalidated near cache
ConfigurationBuilder unbounded = new ConfigurationBuilder();
unbounded.nearCache().mode(NearCacheMode.INVALIDATED).maxEntries(-1);

// Bounded invalidated near cache
ConfigurationBuilder bounded = new ConfigurationBuilder();
bounded.nearCache().mode(NearCacheMode.INVALIDATED).maxEntries(100);
```

Since the configuration is shared by all caches obtained from a single `RemoteCacheManager`, you may not want to enable near-caching for all of them. You can use the `cacheNamePattern` configuration attribute to define a regular expression which matches the names of the caches for which you want near-caching. Caches whose name don't match the regular expression, will not have near-caching enabled.

```
// Bounded invalidated near cache with pattern matching
ConfigurationBuilder bounded = new ConfigurationBuilder();
bounded.nearCache()
  .mode(NearCacheMode.INVALIDATED)
  .maxEntries(100)
  .cacheNamePattern("near.*"); // enable near-cache only for caches whose name starts
with 'near'
```

Near caches work the same way for local caches as they do for clustered caches, but in a clustered cache scenario, if the server node sending the near cache notifications to the Hot Rod client goes down, the Hot Rod client transparently fails over to another node in the cluster, clearing the near cache along the way.

1.1.19. Unsupported methods

Some of the Cache methods are not being supported by the RemoteCache . Calling one of these methods results in an UnsupportedOperationException being thrown. Most of these methods do not make sense on the remote cache (e.g. listener management operations), or correspond to methods that are not supported by local cache as well (e.g. containsValue). Another set of unsupported operations are some of the atomic operations inherited from ConcurrentMap :

```
boolean remove(Object key, Object value);
boolean replace(Object key, Object value);
boolean replace(Object key, Object oldValue, Object value);
```

RemoteCache offers alternative versioned methods for these atomic operations, that are also network friendly, by not sending the whole value object over the network, but a version identifier. See the section on versioned API.

Each one of these unsupported operation is documented in the RemoteCache javadoc.

1.1.20. Return values

There is a set of methods that alter a cached entry and return the previous existing value, e.g.:

```
V remove(Object key);
V put(K key, V value);
```

By default on RemoteCache, these operations return null even if such a previous value exists. This approach reduces the amount of data sent over the network. However, if these return values are needed they can be enforced on a per invocation basis using flags:

```
cache.put("aKey", "initialValue");
assert null == cache.put("aKey", "aValue");
assert "aValue".equals(cache.withFlags(Flag.FORCE_RETURN_VALUE).put("aKey",
   "newValue"));
```

This default behavior can can be changed through force-return-value=true configuration parameter (see configuration section bellow).

Chapter 2. Hot Rod Transactions

You can configure and use Hot Rod clients in JTA Transactions.

To participate in a transaction, the Hot Rod client requires the TransactionManager with which it interacts and whether it participates in the transaction through the Synchronization or XAResource interface.

Transactions are optimistic in that clients acquire write locks on entries during the prepare phase. To avoid data inconsistency, be sure to read about Detecting Conflicts with Transactions.

2.1. Configuring the Server

Caches in the server must also be transactional for clients to participate in JTA Transactions.

The following server configuration is required, otherwise transactions rollback only:

- Isolation level must be `REPEATABLE_READ`.
- Locking mode must be `PESSIMISTIC`. In a future release, `OPTIMISTIC` locking mode will be supported.
- Transaction mode should be `NON_XA` or `NON_DURABLE_XA`. Hot Rod transactions cannot use `FULL_XA` because it degrades performance.

Hot Rod transactions have their own recovery mechanism.

2.2. Configuring Hot Rod Clients

When you create the RemoteCacheManager, you can set the default TransactionManager and TransactionMode that the RemoteCache uses.

The RemoteCacheManager lets you create only one configuration for transactional caches, as in the following example:

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new org
.infinispan.client.hotrod.configuration.ConfigurationBuilder();
//other client configuration parameters
cb.transaction().transactionManagerLookup(GenericTransactionManagerLookup.getInstance(
));
cb.transaction().transactionMode(TransactionMode.NON_XA);
cb.transaction().timeout(1, TimeUnit.MINUTES)
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```

The preceding configuration applies to all instances of a remote cache. If you need to apply different configurations to remote cache instances, you can override the RemoteCache configuration. See Overriding RemoteCacheManager Configuration.

See ConfigurationBuilder Javadoc for documentation on configuration parameters.

You can also configure the Java Hot Rod client with a properties file, as in the following example:

```
infinispan.client.hotrod.transaction.transaction_manager_lookup =
org.infinispan.client.hotrod.transaction.lookup.GenericTransactionManagerLookup
infinispan.client.hotrod.transaction.transaction_mode = NON_XA
infinispan.client.hotrod.transaction.timeout = 60000
```

2.2.1. TransactionManagerLookup Interface

`TransactionManagerLookup` provides an entry point to fetch a TransactionManager.

Available implementations of `TransactionManagerLookup`:

GenericTransactionManagerLookup

A lookup class that locates TransactionManagers running in Java EE application servers. Defaults to the RemoteTransactionManager if it cannot find a TransactionManager.

In most cases, GenericTransactionManagerLookup is suitable. However, you can implement the `TransactionManagerLookup` interface if you need to integrate a custom TransactionManager.

RemoteTransactionManagerLookup

A basic, and volatile, TransactionManager if no other implementation is available. Note that this implementation has significant limitations when handling concurrent transactions and recovery.

2.2.2. Transaction Modes

TransactionMode controls how a RemoteCache interacts with the TransactionManager.

Configure transaction modes on both the Infinispan server and your client application. If clients attempt to perform transactional operations on non-transactional caches, runtime exceptions can occur.

Transaction modes are the same in both the Infinispan configuration and client settings. Use the following modes with your client, see the Infinispan configuration schema for the server:

`NONE`

The RemoteCache does not interact with the TransactionManager. This is the default mode and is non-transactional.

`NON_XA`

The RemoteCache interacts with the TransactionManager via Synchronization.

`NON_DURABLE_XA`

The RemoteCache interacts with the TransactionManager via XAResource. Recovery capabilities

are disabled.

`FULL_XA`
: The RemoteCache interacts with the TransactionManager via XAResource. Recovery capabilities are enabled. Invoke the `XaResource.recover()` method to retrieve transactions to recover.

2.3. Overriding Configuration for Cache Instances

Because RemoteCacheManager does not support different configurations for each cache instance. However, RemoteCacheManager includes the `getCache(String)` method that returns the RemoteCache instances and lets you override some configuration parameters, as follows:

`getCache(String cacheName, TransactionMode transactionMode)`
: Returns a RemoteCache and overrides the configured TransactionMode.

`getCache(String cacheName, boolean forceReturnValue, TransactionMode transactionMode)`
: Same as previous, but can also force return values for write operations.

`getCache(String cacheName, TransactionManager transactionManager)`
: Returns a RemoteCache and overrides the configured TransactionManager.

`getCache(String cacheName, boolean forceReturnValue, TransactionManager transactionManager)`
: Same as previous, but can also force return values for write operations.

`getCache(String cacheName, TransactionMode transactionMode, TransactionManager transactionManager)`
: Returns a RemoteCache and overrides the configured TransactionManager and TransactionMode. Uses the configured values, if `transactionManager` or `transactionMode` is null.

`getCache(String cacheName, boolean forceReturnValue, TransactionMode transactionMode, TransactionManager transactionManager)`
: Same as previous, but can also force return values for write operations.

The `getCache(String)` method returns RemoteCache instances regardless of whether they are transaction or not. RemoteCache includes a `getTransactionManager()` method that returns the TransactionManager that the cache uses. If the RemoteCache is not transactional, the method returns `null`.

2.4. Detecting Conflicts with Transactions

Transactions use the initial values of keys to detect conflicts. For example, "k" has a value of "v" when a transaction begins. During the prepare phase, the transaction fetches "k" from the server to read the value. If the value has changed, the transaction rolls back to avoid a conflict.

Transactions use versions to detect changes instead of checking value equality.

The `forceReturnValue` parameter controls write operations to the RemoteCache and helps avoid conflicts. It has the following values:

- If `true`, the TransactionManager fetches the most recent value from the server before performing write operations. However, the `forceReturnValue` parameter applies only to write operations that access the key for the first time.
- If `false`, the TransactionManager does not fetch the most recent value from the server before performing write operations. Because this setting

This parameter does not affect *conditional* write operations such as `replace` or `putIfAbsent` because they require the most recent value.

The following transactions provide an example where the `forceReturnValue` parameter can prevent conflicting write operations:

Transaction 1 (TX1)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.put("k", "v1");
tm.commit();
```

Transaction 2 (TX2)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.put("k", "v2");
tm.commit();
```

In this example, TX1 and TX2 are executed in parallel. The initial value of "k" is "v".

- If `forceReturnValue = true`, the `cache.put()` operation fetches the value for "k" from the server in both TX1 and TX2. The transaction that acquires the lock for "k" first then commits. The other transaction rolls back during the commit phase because the transaction can detect that "k" has a value other than "v".
- If `forceReturnValue = false`, the `cache.put()` operation does not fetch the value for "k" from the server and returns null. Both TX1 and TX2 can successfully commit, which results in a conflict. This occurs because neither transaction can detect that the initial value of "k" changed.

The following transactions include `cache.get()` operations to read the value for "k" before doing the `cache.put()` operations:

Transaction 1 (TX1)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.get("k");
cache.put("k", "v1");
tm.commit();
```

Transaction 2 (TX2)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.get("k");
cache.put("k", "v2");
tm.commit();
```

In the preceding examples, TX1 and TX2 both read the key so the `forceReturnValue` parameter does not take effect. One transaction commits, the other rolls back. However, the `cache.get()` operation requires an additional server request. If you do not need the return value for the `cache.put()` operation that server request is inefficient.

2.5. Using the Configured Transaction Manager and Transaction Mode

The following example shows how to use the `TransactionManager` and `TransactionMode` that you configure in the `RemoteCacheManager`:

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new org
.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);

RemoteCacheManager rcm = new RemoteCacheManager(cb.build());

//The my-cache instance uses the RemoteCacheManager configuration.
RemoteCache<String, String> cache = rcm.getCache("my-cache");

//Return the transaction manager that the cache uses.
TransactionManager tm = cache.getTransactionManager();

//Perform a simple transaction.
tm.begin();
cache.put("k1", "v1");
System.out.println("K1 value is " + cache.get("k1"));
tm.commit();
```

2.6. Overriding the Transaction Manager

The following example shows how to override `TransactionManager` with the `getCache` method:

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new org
.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);

RemoteCacheManager rcm = new RemoteCacheManager(cb.build());

//Define a custom TransactionManager.
TransactionManager myCustomTM = ...

//Override the TransactionManager for the my-cache instance. Use the default
configuration if null is returned.
RemoteCache<String, String> cache = rcm.getCache("my-cache", null, myCustomTM);

//Perform a simple transaction.
myCustomTM.begin();
cache.put("k1", "v1");
System.out.println("K1 value is " + cache.get("k1"));
myCustomTM.commit();
```

2.7. Overriding the Transaction Mode

The following example shows how to override `TransactionMode` with the `getCache` method:

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new org
.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);

RemoteCacheManager rcm = new RemoteCacheManager(cb.build());

//Override the transaction mode for the my-cache instance.
RemoteCache<String, String> cache = rcm.getCache("my-cache", TransactionMode
.NON_DURABLE_XA, null);

//Return the transaction manager that the cache uses.
TransactionManager tm = cache.getTransactionManager();

//Perform a simple transaction.
tm.begin();
cache.put("k1", "v1");
System.out.println("K1 value is " + cache.get("k1"));
tm.commit();
```

2.7.1. Client Intelligence

Client intelligence refers to mechanisms the `HotRod` protocol provides for clients to locate and send requests to Infinispan servers.

Basic intelligence

Clients do not store any information about Infinispan clusters or key hash values.

Topology-aware

Clients receive and store information about Infinispan clusters. Clients maintain an internal mapping of the cluster topology that changes whenever servers join or leave clusters.

To receive a cluster topology, clients need the address (`IP:HOST`) of at least one Hot Rod server at startup. After the client connects to the server, Infinispan transmits the topology to the client. When servers join or leave the cluster, Infinispan transmits an updated topology to the client.

Distribution-aware

Clients are topology-aware and store consistent hash values for keys.

For example, take a `put(k,v)` operation. The client calculates the hash value for the key so it can locate the exact server on which the data resides. Clients can then connect directly to the owner to dispatch the operation.

The benefit of distribution-aware intelligence is that Infinispan servers do not need to look up values based on key hashes, which uses less resources on the server side. Another benefit is that servers respond to client requests more quickly because it skips additional network roundtrips.

2.7.2. Request Balancing

Clients that use topology-aware intelligence use request balancing for all requests. The default balancing strategy is round-robin, so topology-aware clients always send requests to servers in round-robin order.

For example, `s1`, `s2`, `s3` are servers in a Infinispan cluster. Clients perform request balancing as follows:

```
CacheContainer cacheContainer = new RemoteCacheManager();
Cache<String, String> cache = cacheContainer.getCache();

//client sends put request to s1
cache.put("key1", "aValue");
//client sends put request to s2
cache.put("key2", "aValue");
//client sends get request to s3
String value = cache.get("key1");
//client dispatches to s1 again
cache.remove("key2");
//and so on...
```

Clients that use distribution-aware intelligence use request balancing only for failed requests. When requests fail, distribution-aware clients retry the request on the next available server.

Custom balancing policies

You can implement FailoverRequestBalancingStrategy and specify your class in your `hotrod-client.properties` configuration.

2.7.3. Persistent connections

In order to avoid creating a TCP connection on each request (which is a costly operation), the client keeps a pool of persistent connections to all the available servers and it reuses these connections whenever it is possible. The validity of the connections is checked using an async thread that iterates over the connections in the pool and sends a HotRod ping command to the server. By using this connection validation process the client is being proactive: there's a hight chance for broken connections to be found while being idle in the pool and no on actual request from the application.

The number of connections per server, total number of connections, how long should a connection be kept idle in the pool before being closed - all these (and more) can be configured. Please refer to the javadoc of RemoteCacheManager for a list of all possible configuration elements.

2.7.4. Marshalling data

The Hot Rod client allows one to plug in a custom marshaller for transforming user objects into byte arrays and the other way around. This transformation is needed because of Hot Rod's binary nature - it doesn't know about objects.

The marshaller can be plugged through the "marshaller" configuration element (see Configuration section): the value should be the fully qualified name of a class implementing the Marshaller interface. This is a optional parameter, if not specified it defaults to the ProtoStreamMarshaller

WARNING: If developing your own custom marshaller, take care of potential injection attacks.

To avoid such attacks, make the marshaller verify that any class names read, before instantiating it, is amongst the expected/allowed class names.

The client configuration can be enhanced with a list of regular expressions for classes that are allowed to be read.

WARNING: These checks are opt-in, so if not configured, any class can be read.

In the example below, only classes with fully qualified names containing `Person` or `Employee` would be allowed:

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;

...
ConfigurationBuilder configBuilder = ...
configBuilder.addJavaSerialWhiteList(".*Person.*", ".*Employee.*");
```

ProtoStream

Since version 10.0, the default marshaller for the client is the Protobuf based ProtoStreamMarshaller. Keeping your objects stored in protobuf format has the benefit of being able to consume them with compatible clients written in different languages. To aid with the configuration of the marshaller's SerializationContext, we have extended the client configuration to allow for SerializationContextInitializer(s) to be provided by the user and applied on RemoteCacheManager startup.

SerializationContextInitailizers can be configured as follows:

- Via a comma-separated list of SerializationContextInitializer implementation's in `hotrod-client.properties`

```
infinispan.client.hotrod.context-initializers=org.infinispan.example
.LibraryInitializerImpl,org.infinispan.example.AnotherExampleSciImpl
```

- Programmatically:

```
ConfigurationBuilder builder = new ConfigurationBuilder()
      .addServer().host("localhost").port(hotRodServer.getPort())
      .addContextInitializers(new LibraryInitializerImpl(), new AnotherExampleSciImpl
())
      .build();
RemoteCacheManager rcm = new RemoteCacheManager(builder);
```

2.7.5. Reading data in different data formats

By default, every Hot Rod client operation will use the configured marshaller when reading and writing from the server for both keys and values. See Marshalling Data. Using the DataFormat API, it's possible to decorate remote caches so that all operations can happen with a custom data format.

Using different marshallers for Key and Values

Marshallers for Keys and Values can be overridden at run time. For example, to bypass all serialization in the Hot Rod client and read the byte[] as they are stored in the server:

```
// Existing Remote cache instance
RemoteCache<String, Pojo> remoteCache = ...

// IdentityMarshaller is a no-op marshaller
DataFormat rawKeyAndValues = DataFormat.builder().keyMarshaller(IdentityMarshaller
.INSTANCE).valueMarshaller(IdentityMarshaller.INSTANCE).build();

// Will create a new instance of RemoteCache with the supplied DataFormat
RemoteCache<byte[], byte[]> rawResultsCache = remoteCache.withDataFormat
(rawKeyAndValues);
```

Reading data in different formats

Apart from defining custom key and value marshallers, it's also possible to request/send data in different formats specified by a `org.infinispan.commons.dataconversion.MediaType`:

```
// Existing remote cache using ProtostreamMarshaller
RemoteCache<String, Pojo> protobufCache = ...

// Request values returned as JSON, using the UTF8StringMarshaller that converts
between UTF-8 to String:
DataFormat jsonString = DataFormat.builder().valueType(MediaType.APPLICATION_JSON)
.valueMarshaller(new UTF8StringMarshaller().build();

RemoteCache<String, String> jsonStrCache = remoteCache.withDataFormat(jsonString);

// Alternativelly, it's possible to request JSON values but marshalled/unmarshalled
with a custom value marshaller that returns `org.codehaus.jackson.JsonNode` objects:
DataFormat jsonNode = DataFormat.builder().valueType(MediaType.APPLICATION_JSON)
.valueMarshaller(new CustomJacksonMarshaller().build();

RemoteCache<String, JsonNode> jsonNodeCache = remoteCache.withDataFormat(jsonNode);
```

The data conversions happen in the server, and if it doesn't support converting from the storage format to the request format and vice versa, an error will be returned.

Using different marshallers and formats for the key, with `.keyMarshaller()` and `.keyType()` may interfere with the client intelligence routing mechanism, causing it contact the server that is not the owner of the key during Hot Rod operations. This will not result in errors but can result in extra hops inside the cluster to execute the operation. If performance is critical, make sure to use the keys in the format stored by the server.

2.7.6. Statistics

Various server usage statistics can be obtained through the RemoteCache .stats() method. This returns a ServerStatistics object - please refer to javadoc for details on the available statistics.

2.7.7. Multi-Get Operations

The Java Hot Rod client does not provide multi-get functionality out of the box but clients can build it themselves with the given APIs.

2.7.8. Failover capabilities

Hot Rod clients' capabilities to keep up with topology changes helps with request balancing and more importantly, with the ability to failover operations if one or several of the servers fail.

Some of the conditional operations mentioned above, including `putIfAbsent`, `replace` with and without version, and conditional `remove` have strict method return guarantees, as well as those operations where returning the previous value is forced.

In spite of failures, these methods return values need to be guaranteed, and in order to do so, it's

necessary that these methods are not applied partially in the cluster in the event of failure. For example, imagine a `replace()` operation called in a server for key=k1 with `Flag.FORCE_RETURN_VALUE`, whose current value is `A` and the replace wants to set it to `B`. If the replace fails, it could happen that some servers contain `B` and others contain `A`, and during the failover, the original `replace()` could end up returning `B`, if the replace failovers to a node where `B` is set, or could end up returning `A`.

To avoid this kind of situations, whenever Java Hot Rod client users want to use conditional operations, or operations whose previous value is required, it's important that the cache is configured to be transactional in order to avoid incorrect conditional operations or return values.

2.7.9. Site Cluster Failover

On top of the in-cluster failover, Hot Rod clients are also able to failover to different clusters, which could be represented as an independent site.

The way site cluster failover works is that if all the main cluster nodes are not available, the client checks to see if any other clusters have been defined in which cases it tries to failover to the alternative cluster. If the failover succeeds, the client will remain connected to the alternative cluster until this becomes unavailable, in which case it'll try any other clusters defined, and ultimately, it'll try the original server settings.

To configure a cluster in the Hot Rod client, one host/port pair details must be provided for each of the clusters configured. For example:

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb
      = new org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.addCluster().addClusterNode("remote-cluster-host", 11222);
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```

Remember that regardless of the cluster definitions, the initial server(s) configuration must be provided unless the initial servers can be resolved using the default server host and port details.

2.7.10. Manual Site Cluster Switch

As well as supporting automatic site cluster failover, Java Hot Rod clients can also switch between site clusters manually by calling RemoteCacheManager's `switchToCluster(clusterName)` and `switchToDefaultCluster()`.

Using `switchToCluster(clusterName)`, users can force a client to switch to one of the clusters pre-defined in the Hot Rod client configuration. To switch to the initial servers defined in the client configuration, call `switchToDefaultCluster()`.

2.7.11. Monitoring the Hot Rod client

The Hot Rod client can be monitored and managed via JMX. By enabling statistics, an MBean will be registered for the `RemoteCacheManager` as well as for each `RemoteCache` obtained through it. Through these MBeans it is possible to obtain statistics about remote and near-cache hits/misses and

connection pool usage.

2.7.12. Concurrent Updates

Data structures, such as Infinispan Cache , that are accessed and modified concurrently can suffer from data consistency issues unless there're mechanisms to guarantee data correctness. Infinispan Cache, since it implements ConcurrentMap , provides operations such as conditional replace , putIfAbsent , and conditional remove to its clients in order to guarantee data correctness. It even allows clients to operate against cache instances within JTA transactions, hence providing the necessary data consistency guarantees.

However, when it comes to Hot Rod protocol backed servers, clients do not yet have the ability to start remote transactions but they can call instead versioned operations to mimic the conditional methods provided by the embedded Infinispan cache instance API. Let's look at a real example to understand how it works.

Data Consistency Problem

Imagine you have two ATMs that connect using Hot Rod to a bank where an account's balance is stored. Two closely followed operations to retrieve the latest balance could return 500 CHF (swiss francs) as shown below:

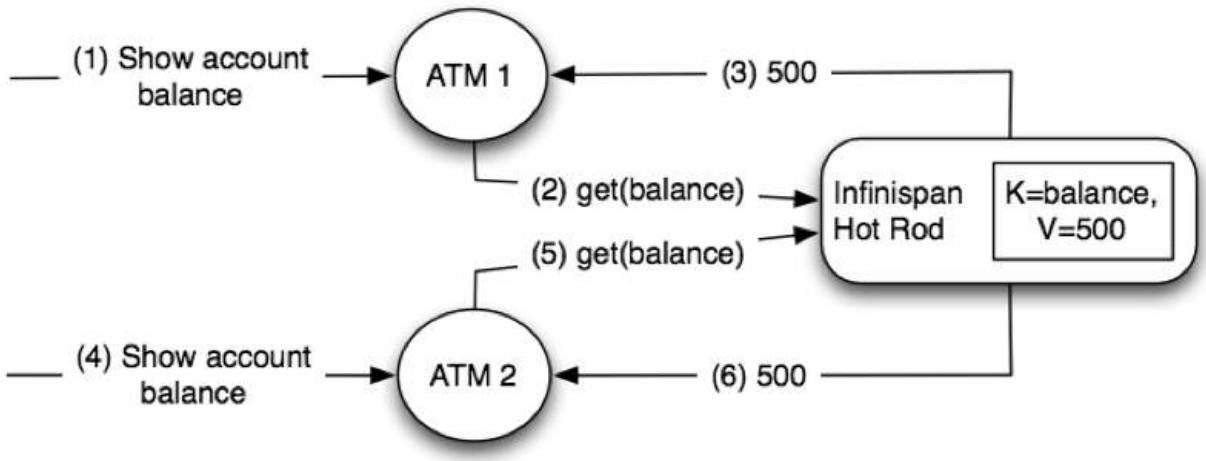
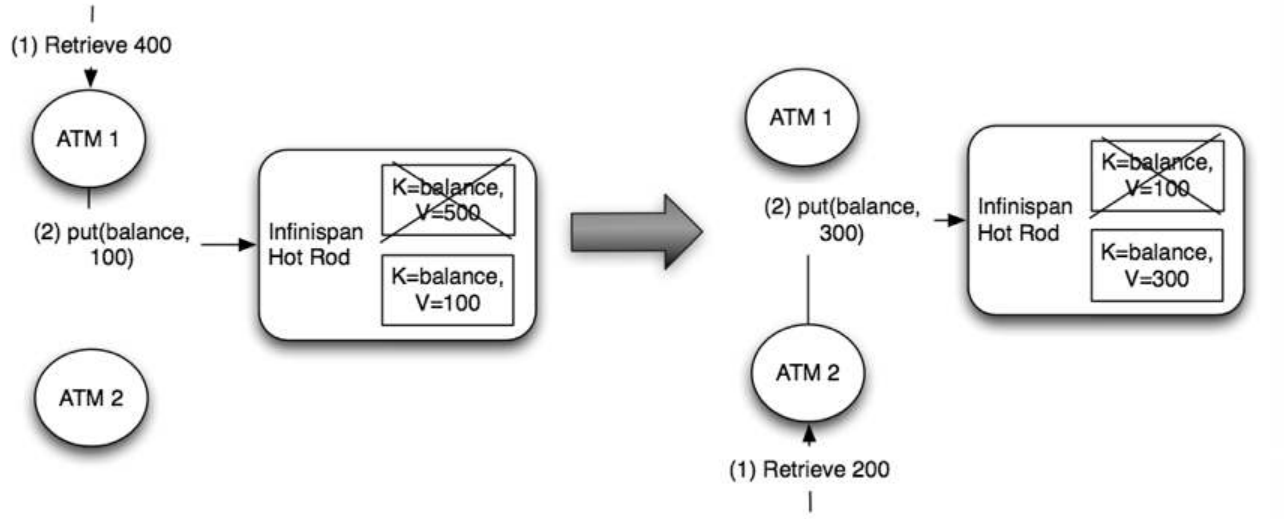


Figure 2. Concurrent readers

Next a customer connects to the first ATM and requests 400 CHF to be retrieved. Based on the last value read, the ATM could calculate what the new balance is, which is 100 CHF, and request a put with this new value. Let's imagine now that around the same time another customer connects to the ATM and requests 200 CHF to be retrieved. Let's assume that the ATM thinks it has the latest balance and based on its calculations it sets the new balance to 300 CHF:



Obviously, this would be wrong. Two concurrent updates have resulted in an incorrect account balance. The second update should not have been allowed since the balance the second ATM had was incorrect. Even if the ATM would have retrieved the balance before calculating the new balance, someone could have updated between the new balance being retrieved and the update. Before finding out how to solve this issue in a client-server scenario with Hot Rod, let's look at how this is solved when Infinispan clients run in peer-to-peer mode where clients and Infinispan live within the same JVM.

Embedded-mode Solution

If the ATM and the Infinispan instance storing the bank account lived in the same JVM, the ATM could use the conditional replace API referred at the beginning of this article. So, it could send the previous known value to verify whether it has changed since it was last read. By doing so, the first operation could double check that the balance is still 500 CHF when it was to update to 100 CHF. Now, when the second operation comes, the current balance would not be 500 CHF any more and hence the conditional replace call would fail, hence avoiding data consistency issues:

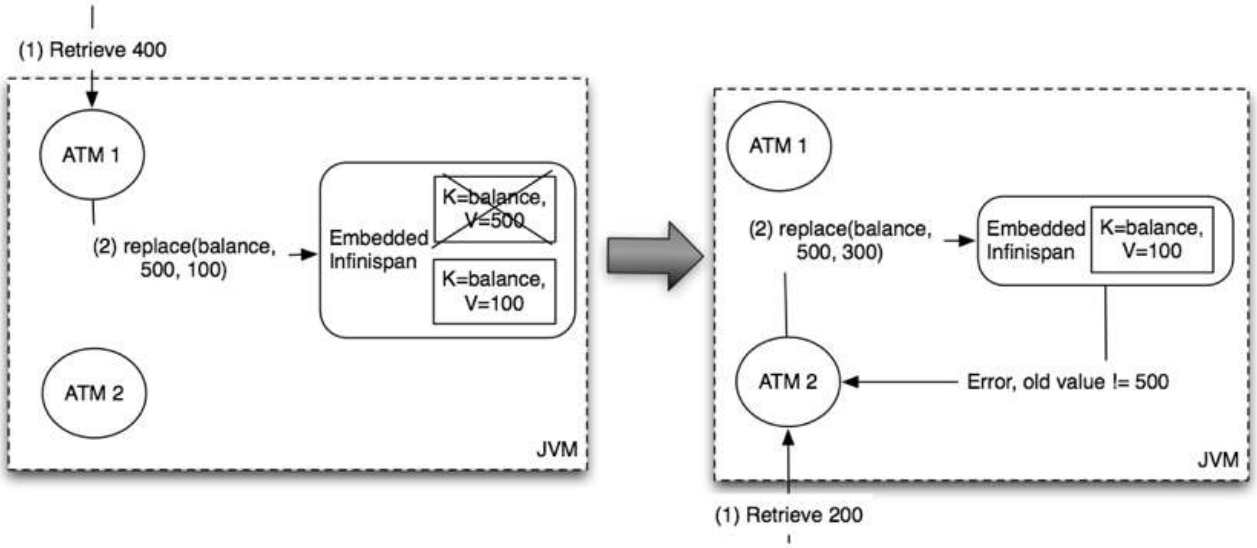


Figure 3. P2P solution

Client-Server Solution

In theory, Hot Rod could use the same p2p solution but sending the previous value would be not practical. In this example, the previous value is just an integer but the value could be a lot bigger and hence forcing clients to send it to the server would be rather wasteful. Instead, Hot Rod offers versioned operations to deal with this situation.

Basically, together with each key/value pair, Hot Rod stores a version number which uniquely identifies each modification. So, using an operation called getVersioned or getWithVersion, clients can retrieve not only the value associated with a key, but also the current version. So, if we look at the previous example once again, the ATMs could call getVersioned and get the balance's version:

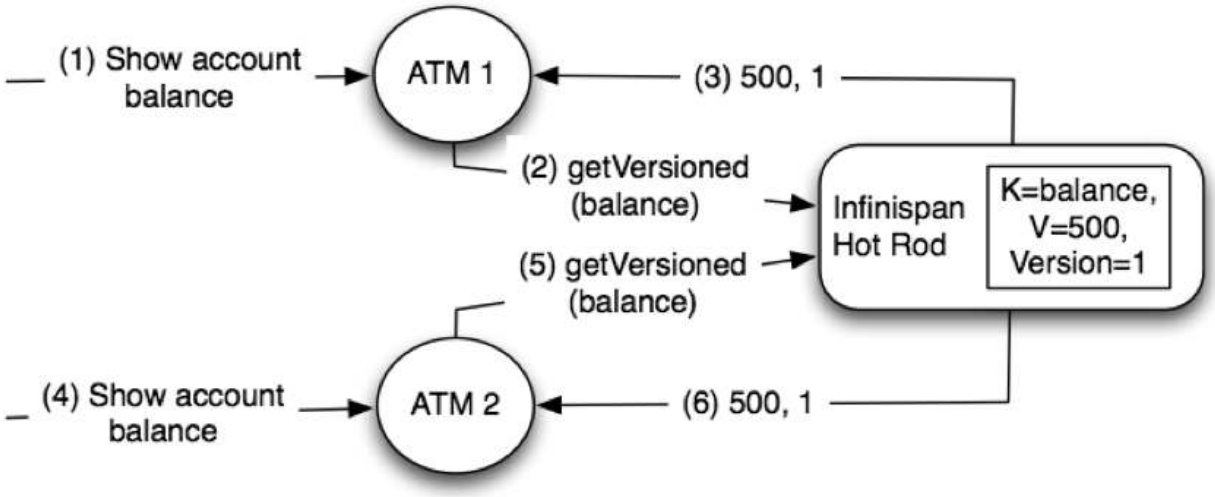


Figure 4. Get versioned

When the ATMs wanted to modify the balance, instead of just calling put, they could call replaceIfUnmodified operation passing the latest version number of which the clients are aware of. The operation will only succeed if the version passed matches the version in the server. So, the first modification by the ATM would be allowed since the client passes 1 as version and the server side version for the balance is also 1. On the other hand, the second ATM would not be able to make the modification because after the first ATMs modification the version would have been incremented to 2, and now the passed version (1) and the server side version (2) would not match:

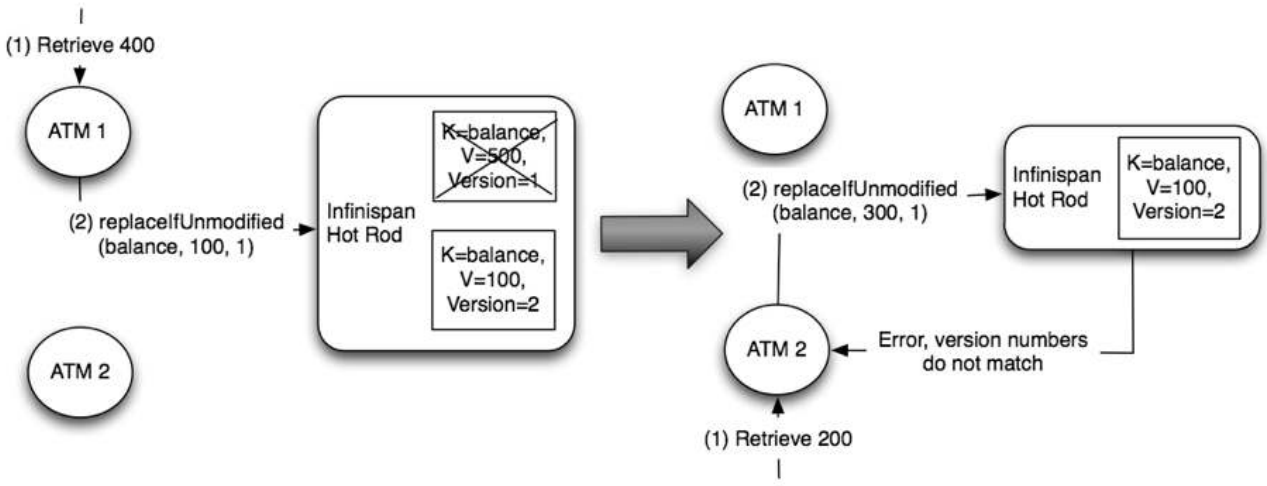


Figure 5. Replace if versions match