

Hot Rod Java Client Guide

Table of Contents

1. Hot Rod Java Clients	1
1.1. Hot Rod Protocol	1
1.2. Downloading the Hot Rod Java Client	1
2. Configuring Hot Rod Java Clients	2
2.1. Programmatically Configuring Hot Rod Java Clients	2
2.2. Configuring Hot Rod Java Client Property Files	2
2.3. Client Intelligence	3
2.3.1. Request Balancing	4
2.3.2. Client Failover	5
2.4. Configuring Authentication Mechanisms for Hot Rod Clients	5
2.4.1. Hot Rod Endpoint Authentication Mechanisms	10
2.4.2. Creating GSSAPI Login Contexts	11
2.5. Configuring Hot Rod Client Encryption	11
2.6. Monitoring Hot Rod Clients via JMX	13
2.7. Defining Infinispan Clusters in Client Configuration	13
2.7.1. Manually Switching Infinispan Clusters	14
2.8. Configuring Connection Pools	14
2.9. Configuring Custom Marshallers	15
2.10. Configuring SerializationContextInitializer Implementations	15
2.11. Configuring Hot Rod Client Data Formats	16
2.12. Configuring Near Caching	17
3. Hot Rod Client API	20
3.1. Basic API	20
3.2. RemoteCache Interface	20
3.3. Remote Iterator	21
3.4. Versioned API	23
3.5. Streaming API	24
3.6. Creating Event Listeners	25
3.6.1. Removing Event Listeners	27
3.6.2. Filtering Events	27
3.6.3. Skipping Notifications	29
3.6.4. Customizing Events	30
3.6.5. Filter and Custom Events	33
3.6.6. Event Marshalling	35
3.6.7. Listener State Handling	36
3.6.8. Listener Failure Handling	36
3.7. Hot Rod Java Client Transactions	37
3.7.1. Configuring the Server	37

3.7.2. Configuring Hot Rod Clients	37
3.7.3. Transaction Modes	38
3.7.4. Overriding Configuration for Cache Instances	39
3.7.5. Detecting Conflicts with Transactions	39
3.7.6. Using the Configured Transaction Manager and Transaction Mode	41
3.7.7. Overriding the Transaction Manager	42
3.7.8. Overriding the Transaction Mode.....	43
3.8. Concurrent Updates	43
3.8.1. Data Consistency Problem	43

Chapter 1. Hot Rod Java Clients

Access Infinispan remotely through the Hot Rod Java client API.

1.1. Hot Rod Protocol

Hot Rod is a binary TCP protocol that Infinispan offers high-performance client-server interactions with the following capabilities:

- Load balancing. Hot Rod clients can send requests across Infinispan clusters using different strategies to increase reliability.
- Failover. Hot Rod clients can monitor Infinispan cluster topology changes and automatically switch to available nodes to ensure requests never go to offline nodes.
- Efficient data location. Hot Rod clients can find key owners and make requests directly to those nodes, which reduces latency.

1.2. Downloading the Hot Rod Java Client

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.

Chapter 2. Configuring Hot Rod Java Clients

2.1. 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](#)

2.2. 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());
```

```
# 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](#)

2.3. Client Intelligence

Hot Rod client intelligence refers to mechanisms for locating Infinispan servers to efficiently route requests.

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.

2.3.1. 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.

2.3.2. Client Failover

Hot Rod clients can automatically failover when Infinispan cluster topologies change. For instance, Hot Rod clients that are topology-aware can detect when one or more Infinispan servers fail.

In addition to failover between clustered Infinispan servers, Hot Rod clients can failover between Infinispan clusters.

For example, you have a Infinispan cluster running in New York (**NYC**) and another cluster running in London (**LON**). Clients sending requests to **NYC** detect that no nodes are available so they switch to the cluster in **LON**. Clients then maintain connections to **LON** until you manually switch clusters or failover happens again.

Transactional Caches with Failover

Conditional operations, such as `putIfAbsent()`, `replace()`, `remove()`, have strict method return guarantees. Likewise, some operations can require previous values to be returned.

Even though Hot Rod clients can failover, you should use transactional caches to ensure that operations do not partially complete and leave conflicting entries on different nodes.

2.4. Configuring Authentication Mechanisms for Hot Rod Clients

Infinispan servers use different mechanisms to authenticate Hot Rod client connections.

Procedure

- Specify the authentication mechanisms that Infinispan server uses with the `saslMechanism()` method from the `SecurityConfigurationBuilder` class.

SCRAM

```
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

```
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

```
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

```
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");
```

OAUTHBEARER authentication with TokenCallbackHandler

You can configure clients with a `TokenCallbackHandler` to refresh OAuth2 tokens before they expire, as in the following example:

```
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

```
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");
```

```

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.

```

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")
                .saslmMechanism("DIGEST-MD5")
                .callbackHandler(new MyCallbackHandler("myuser", "default", "qwer1234!"
                    .toCharArray()));
remoteCacheManager=new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache=remoteCacheManager.getCache("secured");

```

2.4.1. Hot Rod Endpoint Authentication Mechanisms

Infinispan supports the following SASL authentications mechanisms with the Hot Rod connector:

Authentication mechanism	Description	Related details
PLAIN	Uses credentials in plain-text format. You should use PLAIN authentication with encrypted connections only.	Similar to the Basic HTTP mechanism.
DIGEST-*	Uses hashing algorithms and nonce values. Hot Rod connectors support DIGEST-MD5 , DIGEST-SHA , DIGEST-SHA-256 , DIGEST-SHA-384 , and DIGEST-SHA-512 hashing algorithms, in order of strength.	Similar to the Digest HTTP mechanism.
SCRAM-*	Uses <i>salt</i> values in addition to hashing algorithms and nonce values. Hot Rod connectors support SCRAM-SHA , SCRAM-SHA-256 , SCRAM-SHA-384 , and SCRAM-SHA-512 hashing algorithms, in order of strength.	Similar to the Digest HTTP mechanism.
GSSAPI	Uses Kerberos tickets and requires a Kerberos Domain Controller. You must add a corresponding kerberos server identity in the realm configuration. In most cases, you also specify an ldap-realm to provide user membership information.	Similar to the SPNEGO HTTP mechanism.
GS2-KRB5	Uses Kerberos tickets and requires a Kerberos Domain Controller. You must add a corresponding kerberos server identity in the realm configuration. In most cases, you also specify an ldap-realm to provide user membership information.	Similar to the SPNEGO HTTP mechanism.
EXTERNAL	Uses client certificates.	Similar to the CLIENT_CERT HTTP mechanism.
OAuthBEARER	Uses OAuth tokens and requires a token-realm configuration.	Similar to the BEARER_TOKEN HTTP mechanism.

2.4.2. Creating GSSAPI Login Contexts

To use the GSSAPI mechanism, you must create a *LoginContext* so your Hot Rod client can obtain a Ticket Granting Ticket (TGT).

Procedure

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**.

2.5. Configuring Hot Rod Client Encryption

Infinispan servers that use SSL/TLS encryption present Hot Rod clients with certificates so they can establish trust and negotiate secure connections.

To verify server-issued certificates, Hot Rod clients require part of the TLS certificate chain. For example, the following image shows a certificate authority (CA), named "CA", that has issued a certificate for a server named "HotRodServer":

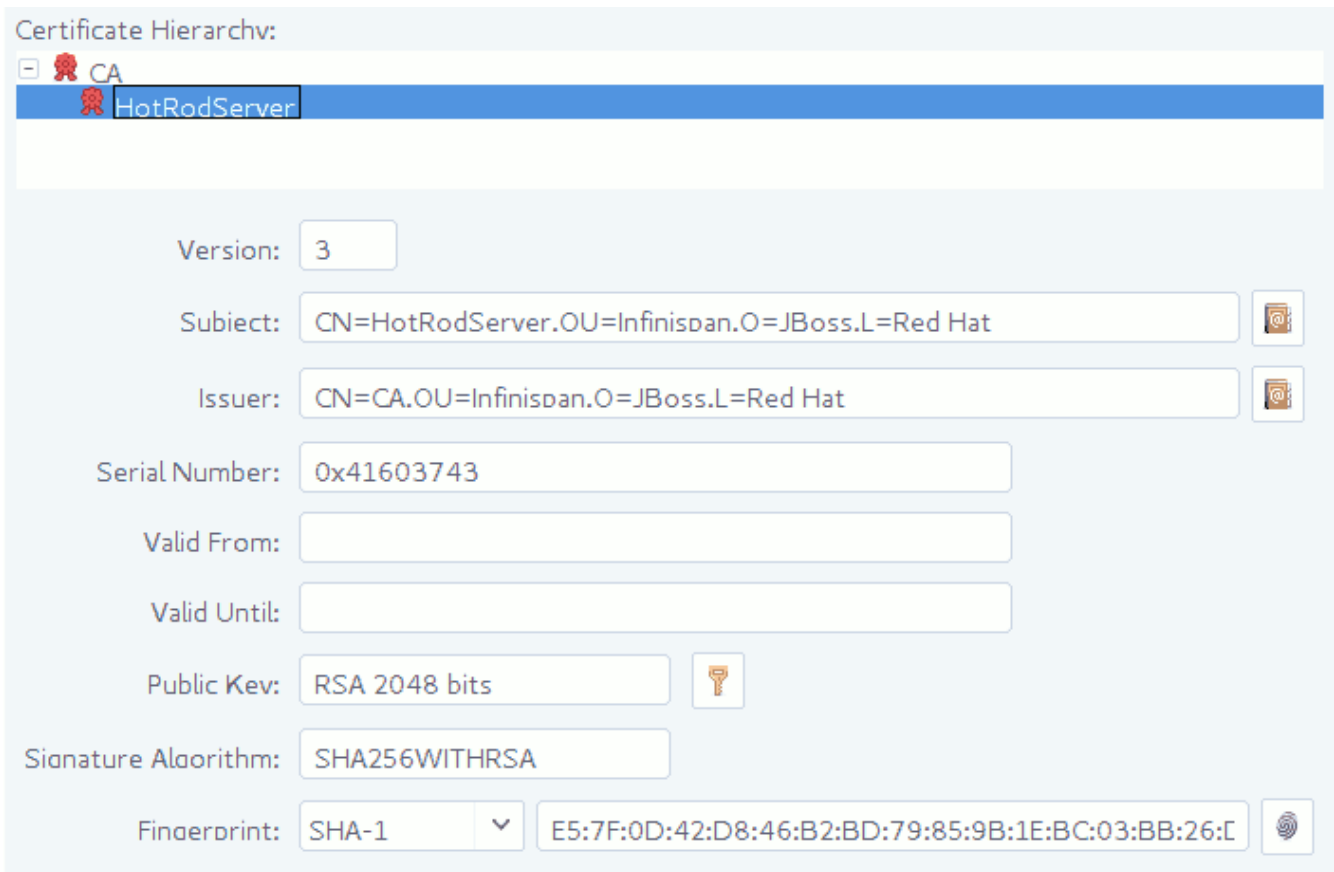


Figure 1. Certificate chain

Procedure

1. Create a Java keystore with part of the server certificate chain. In most cases you should use the public certificate for the CA.
2. Specify the keystore as a *TrustStore* in the client configuration with the `SslConfigurationBuilder` class.

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .security()
        .ssl()
            // Server SNI hostname.
            .sniHostName("myservername") ①
            // Server certificate keystore.
            .trustStoreFileName("/path/to/truststore") ②
            .trustStorePassword("truststorepassword".toCharArray())
            // Client certificate keystore.
            .keyStoreFileName("/path/to/client/keystore") ③
            .keyStorePassword("keystorepassword".toCharArray());
RemoteCache<String, String> cache=remoteCacheManager.getCache("secured");
```

- ① specifies the Infinispan servers Server Name Indication (SNI). This configuration is required if Infinispan servers present multiple identities, which are typically bound to separate cache

containers.

- ② specifies the keystore that contains the server certificate.



Specify a path that contains certificates in PEM format and Hot Rod clients automatically generate trust stores.

Use `.trustStorePath("/path/to/certificate")`.

- ③ specifies the keystore that contains the client certificate. This configuration is required if Hot Rod client must present certificates to Infinispan servers.

2.6. Monitoring Hot Rod Clients via JMX

Enable Hot Rod client statistics via JMX to register an MBean for the `RemoteCacheManager` and each `RemoteCache`.

Hot Rod client statistics include remote and near-cache hits and misses as well as connection pool usage.

Procedure

- Use the `StatisticsConfigurationBuilder` class to enable and configure Hot Rod client statistics.

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .statistics()
    .enable() ①
    .jmxDomain("org.infinispan") ②
    .addServer()
    .host("127.0.0.1")
    .port(11222);
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
```

- ① enables client statistics.

- ② sets the JMX domain name to which MBeans are exposed.

2.7. Defining Infinispan Clusters in Client Configuration

Provide the locations of Infinispan clusters in Hot Rod client configuration.

Procedure

- Provide at least one Infinispan cluster hostname and port with the `ClusterConfigurationBuilder` class.


```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addCluster()
    .addClusterNode("cluster-hostname", 11222); ①
    .addServer() ②
    .host("127.0.0.1")
    .port(11222);
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
```

① sets the connection details for a node in a Infinispan cluster.

② defines the default Infinispan server for Hot Rod client requests.

2.7.1. Manually Switching Infinispan Clusters

Manually switch Hot Rod Java client connections between Infinispan clusters.

Procedure

- Call one of the following methods in the `RemoteCacheManager` class:

`switchToCluster(clusterName)` switches to a specific cluster defined in the client configuration.

`switchToDefaultCluster()` switches to the default cluster defined in the client configuration.

Reference

[RemoteCacheManager](#)

2.8. Configuring Connection Pools

Hot Rod Java clients keep pools of persistent connections to Infinispan servers to reuse TCP connections instead of creating them on each request.

Clients use asynchronous threads that check the validity of connections by iterating over the connection pool and sending pings to Infinispan servers. This improves performance by finding broken connections while they are idle in the pool, rather than on application requests.

Procedure

- Configure Hot Rod client connection pool settings with the `ConnectionPoolConfigurationBuilder` class.

```

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
    .host("127.0.0.1")
    .port(11222);
    .connectionPool() ①
    .maxActive(10) ②
    .minIdle(20); ③
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());

```

- ① configures client connection pools.
- ② sets the maximum number of active connections per server.
- ③ sets the minimum number of idle connections that must be available per server.

2.9. Configuring Custom Marshallers

Hot Rod is a binary TCP protocol, which requires you to transform between user objects and byte arrays.

By default, Infinispan uses a ProtoStream marshaller. However you can configure Hot Rod clients to use custom marshallers.

Procedure

1. Implement the `org.infinispan.commons.marshall.Marshaller` interface.
2. Specify the fully qualified name of your class in your Hot Rod client configuration.
3. Avoid potential injection attacks with custom marshallers by adding a whitelist of class names that can be read.

In the following example, only classes with fully qualified names that contain `Person` or `Employee` are allowed:

```

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.marshaller("org.infinispan.example.marshall.CustomMarshaller");
    .addJavaSerialWhitelist(".*Person.*", ".*Employee.*");
    ...

```

Reference

[Marshaller](#)

2.10. Configuring SerializationContextInitializer Implementations

You can add implementations of the ProtoStream `SerializationContextInitializer` interface to Hot Rod client configurations.

Procedure

- Configure `SerializationContextInitializer` as follows:

`hotrod-client.properties`

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

Programmatic configuration

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
    .host("127.0.0.1")
    .port(11222)
    .addContextInitializers(new LibraryInitializerImpl(), new AnotherExampleSciImpl())
    .build();
RemoteCacheManager rcm = new RemoteCacheManager(builder);
```

2.11. Configuring Hot Rod Client Data Formats

By default, Hot Rod client operations use the configured marshaller when reading and writing from Infinispan servers for both keys and values.

However, the `DataFormat` API lets you decorate remote caches so that all operations can happen with custom data formats.

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);
```



Using different marshallers and formats for keys, with `keyMarshaller()` and `keyType()` methods might interfere with the client intelligence routing mechanism and cause extra hops within the Infinispan cluster to perform the operation. If performance is critical, you should use keys in the format stored by the server.

Request and send data in different formats specified by a `org.infinispan.commons.dataconversion.MediaType` as follows:

```
// 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);

// Alternatively, 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);
```

In the preceding example, data conversion happens in the {brandname} server. Infinispan returns an error if it does not support conversion to and from a storage format.

Reference

[DataFormat](#)

2.12. Configuring Near Caching

Hot Rod Java clients can keep local caches that store recently used data, which significantly increases performance of `get()` and `getVersioned()` operations because the data is local to the client.

When you enable near caching with Hot Rod Java clients, calls to `get()` or `getVersioned()` calls populate the near cache when entries are retrieved from servers. When entries are updated or removed on the server-side, entries in the near cache are invalidated. If keys are requested after they are invalidated, clients must fetch the keys from the server again.

You can also configure the size of near caches by specifying the maximum number of entries that can reside in the near cache. When the maximum is reached, near-cached entries are evicted.

Near cache considerations

- near caches are cleared when clients failover to different servers when using clustered cache modes.
- Do not use maximum idle expiration with near caches because near-cache reads do not propagate the last access time for entries.
- You should always configure the maximum number of entries that can reside in the near cache. Unbounded near caches require you to keep the size of the near cache within the boundaries of

the client JVM.

- Near cache invalidation messages can degrade performance of write operations

Procedure

1. Set the near cache mode to **INVALIDATED** in the client configuration.
2. Define the size of the near cache by specifying the maximum number of entries.

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

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

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



Use `cacheNamePattern()` to define a regular expression to match caches for which you want to enable near caching. Because the configuration is shared for all caches from the `RemoteCacheManager`, this lets you enable near-caching for only the caches that match the regular expression.

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

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.

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 be changed through `force-return-value=true` configuration parameter.

Chapter 3. Hot Rod Client API

Infinispan Hot Rod client API provides interfaces for creating caches remotely, manipulating data, monitoring the topology of clustered caches, and more.

3.1. Basic API

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

3.2. RemoteCache Interface

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 cautioning with this as a large number of keys can and will cause `OutOfMemoryError` in the client.

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

3.3. 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 don't 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.

3.4. 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 below 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.

3.5. 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`).

3.6. 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());

```

3.6.1. Removing Event Listeners

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

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

3.6.2. 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 be marshallable when they are deployed in a cluster so that the filtering can happen right where the event is generated, even if the event 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.

3.6.3. 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");
```

3.6.4. 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 be marshallable when they are deployed in a cluster, so that the conversion can happen right where the event is generated, even if the event 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.

3.6.5. 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);
```

3.6.6. 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 than 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.

3.6.7. 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.

3.6.8. 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 be 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.

3.7. Hot Rod Java Client 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](#).

3.7.1. 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.

3.7.2. 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
```

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.

3.7.3. 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.

3.7.4. 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`.

3.7.5. 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.

3.7.6. 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();
```

3.7.7. 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();
```

3.7.8. 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();
```

3.8. 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.

3.8.1. 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:

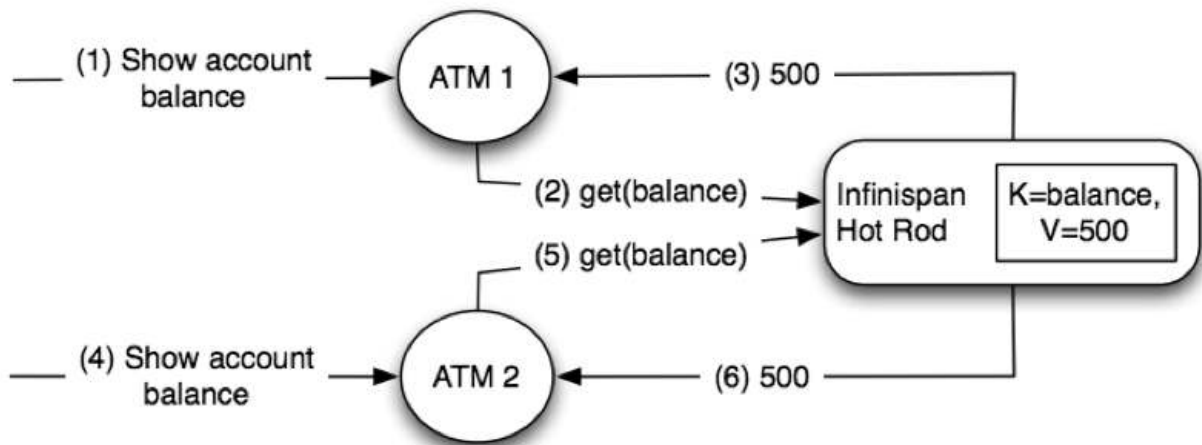
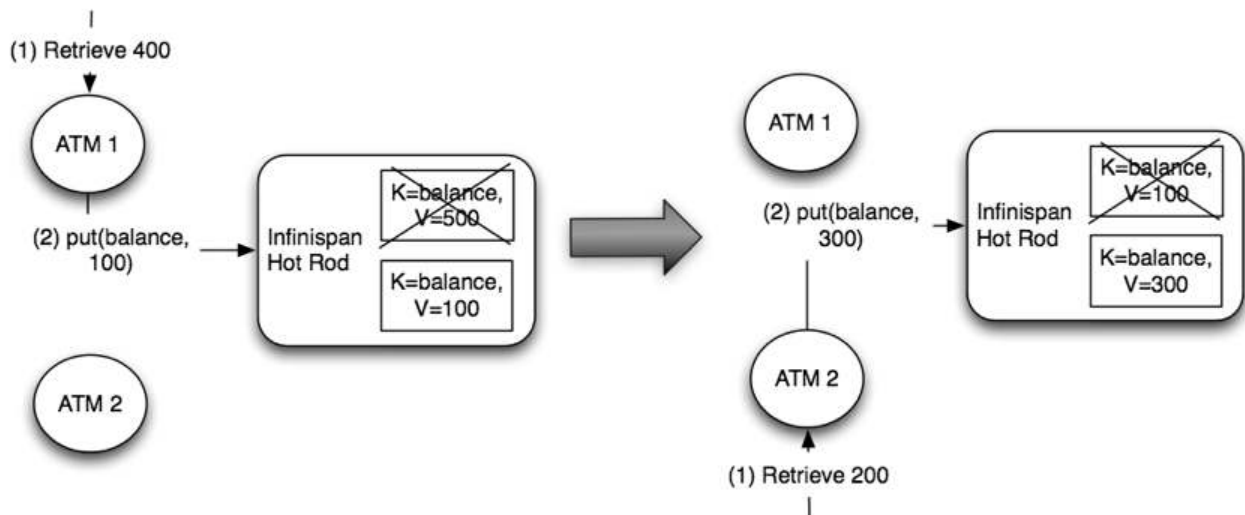


Figure 2. Concurrent readers

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:

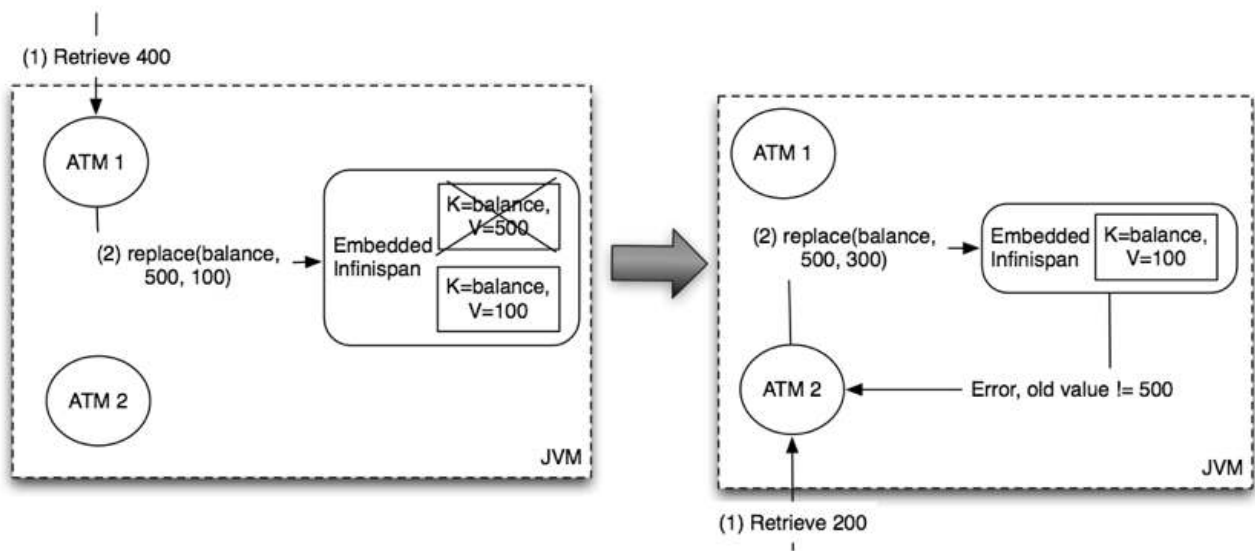


Figure 3. P2P solution

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:

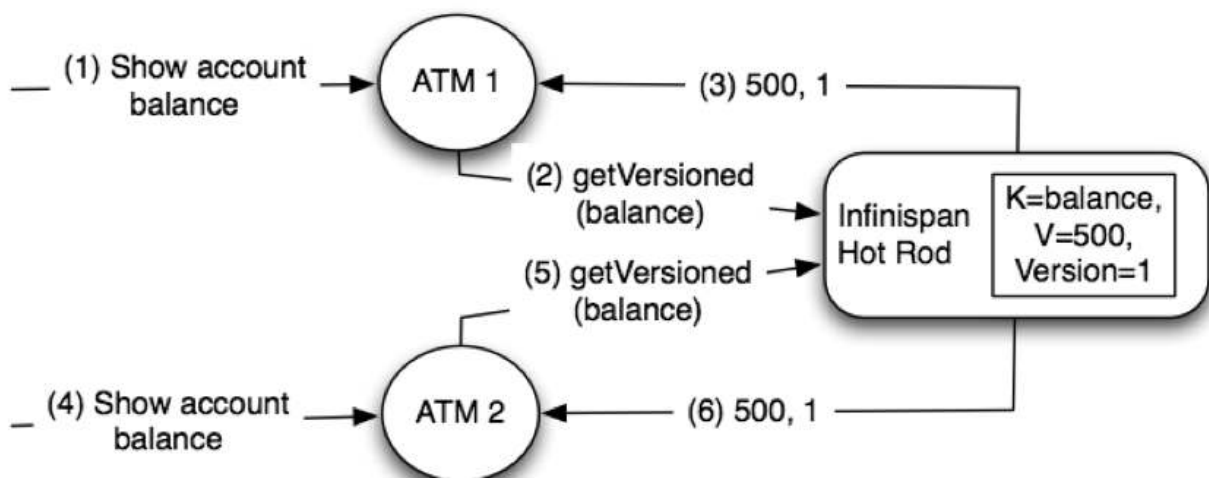


Figure 4. Get versioned

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 ATM's modification the version would have been incremented to 2, and now the passed version (1) and the server side version (2) would not match:

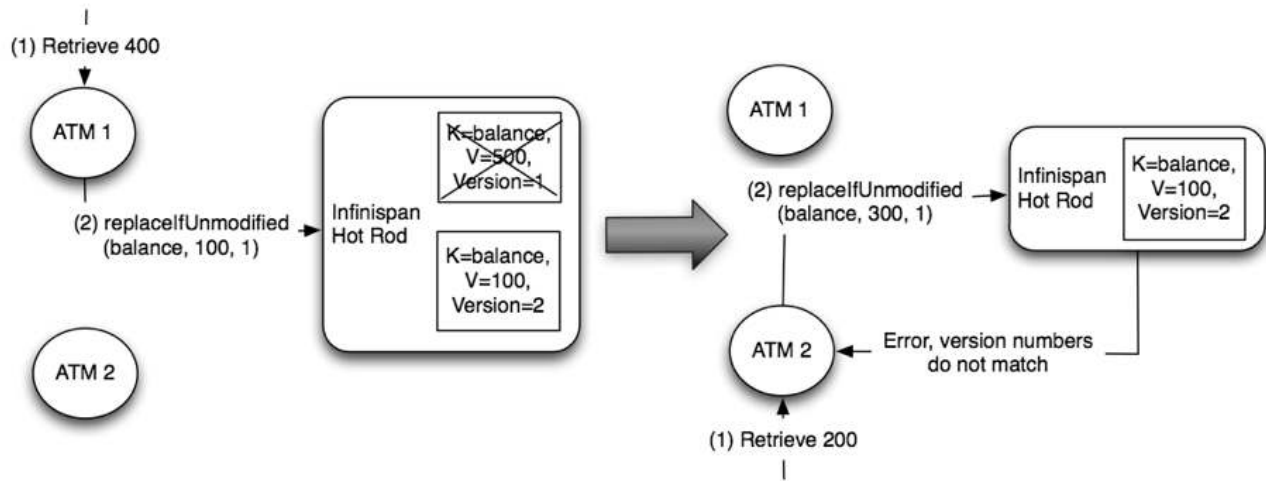


Figure 5. Replace if versions match