software AG

Cross-Language Clients User Guide

Innovation Release

Version 4.3.5

April 2018

TERRACOTTA

Table of Contents

1   About Cross-Language Clients

What are Cross-Language Clients?

BigMemory Cross-Language Clients provide access to BigMemory data from multiple client platforms:

- Java

- .NET C#

- C++

Interoperability between client platforms means that a C# application can access data created by a Java application, or a Java application can access data created by C++ application.

Now .NET and C++ applications can take full advantage of BigMemory's scalability and performance.
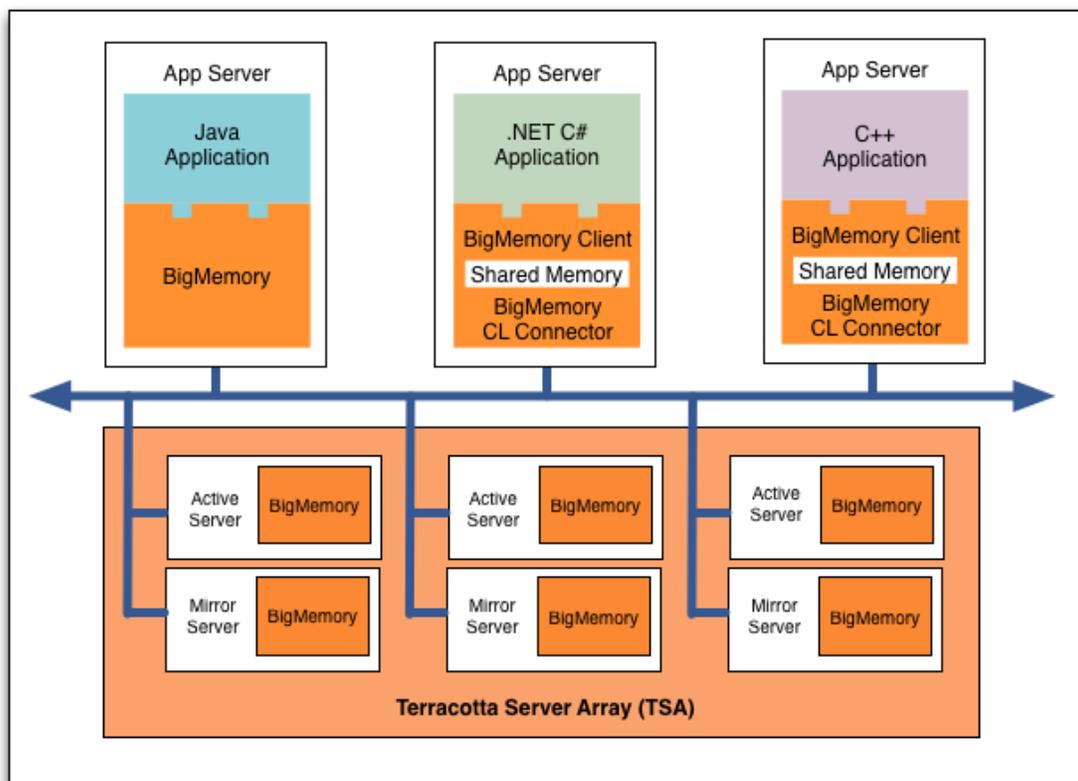
The BigMemory (.NET or C++) client is a library installed on a target platform, with APIs for communication and data transport to BigMemory via the Cross Language (CL) Connector. Connection to the CL Connector is via Nirvana Sockets or Shared Memory transport (Universal Messaging). The CL Connector runs as a JVM process communicating to BigMemory.

2 BigMemory .NET Client

Concepts and Architecture

The BigMemory .NET Client and Cross-Language Connector give you interoperability between your C# application and other platforms by providing access to BigMemory in-memory data. The client and connector may also be used in a single-language environment for fast, reliable access to BigMemory in-memory data.

Interoperable Access to In-Memory Data

BigMemory Cross-Language Clients provide access to data from multiple platforms. Supported platforms can interoperate using the same data. For example, a Java application can read data added by a C# client, and vice versa. The BigMemory Clients also allow searching of the BigMemory data. There is no additional piece of infrastructure required.

A Java VM is set up on the same hardware that hosts the other platform's code. This VM will host the Cross-Language Connector and the BigMemory instance. The Cross-Language Connector runs as a JVM process that talks to the BigMemory instance and opens the communication channel between other processes and BigMemory. The Cross-Language Connector runs as any other Terracotta client (L1) from the cluster perspective, but it allows other processes to connect to it using a well-defined protocol and API.

The BigMemory (.Net or C++) Client connects and accesses data stored in BigMemory. The BigMemory Client is a library which is installed on the target platform in the appropriate path. Within the target application, the API is used to talk to the VM.

Figure 1 below illustrates a topology where multiple clients share one Cross-Language Connector. The BigMemory Client is added to the application, and then the Cross-Language Connector provides access to the BigMemory in-memory data.

Figure 1. Cross-Language Topology, Example 1

Figure 2 below illustrates a topology with multiple Cross-Language Connectors. Hosting the CL Connector on the same machine as your application may have the advantage of keeping your data closer and reducing latency. Each CL Connector requires its own CacheManager, so this topology may be a good approach for keeping discrete data sets available to different applications.

Figure 2. Cross-Language Topology, Example 2

Factors to consider when designing your topology include: your data access patterns, how data is shared among your applications, the amount of memory you have, and the amount of data that you want keep in memory.

Figure 3 below illustrates the flow of data through the Cross-Language Connector.

Figure 3. BigMemory Cross-Language Clients' Data Flow

BigMemory Clients' Data Flow

1. Data from a C# or C++ client application can be stored in BigMemory using the BigMemory Client. A put from a BigMemory Client is converted into platform-neutral format (PNF). It is then transported to the Cross-Language Connector and put into BigMemory, either directly or after serialization, depending upon the use case.

2. Data can be retrieved from BigMemory by the same client or by a different client. A get from the client retrieves the serialized object from BigMemory and deserializes it, making the Domain Object available to the client.

3. You may also have a Java client, which can store and access BigMemory data in either of these ways:

- If the BigMemory data is primarily in platform-neutral format (PNF), use the Ehcache API through a CacheSerializer that lazily serializes/deserializes the data for use with your Java application.

- If the BigMemory data is primarily in Java format, use the Ehcache API directly from Java. (F) In this case, the Cross-Language Connector for .NET or C++ clients should include a Java CacheSerializer.

Single-Language Environments

The BigMemory Cross-Language Clients can also be used for fast and reliable access to in-memory data that has been stored in BigMemory in the native format of your application. This type of installation can be used in homogenous .NET or C++ environments, where interoperability with data in other formats is not required.

Figure 4 below illustrates the flow of data through the Cross-Language Connector in a single-language environment.

Figure 4. Single-Language Data Flow

BigMemory Clients' Data Flow

A. .NET Application with BigMemory Client

B. .NET Application with BigMemory Client

put

get

Domain Object Serialization

Domain Object Deserialization

Cross-Language (CL) Connector

BigMemory
in-memory data
(data in .NET format)
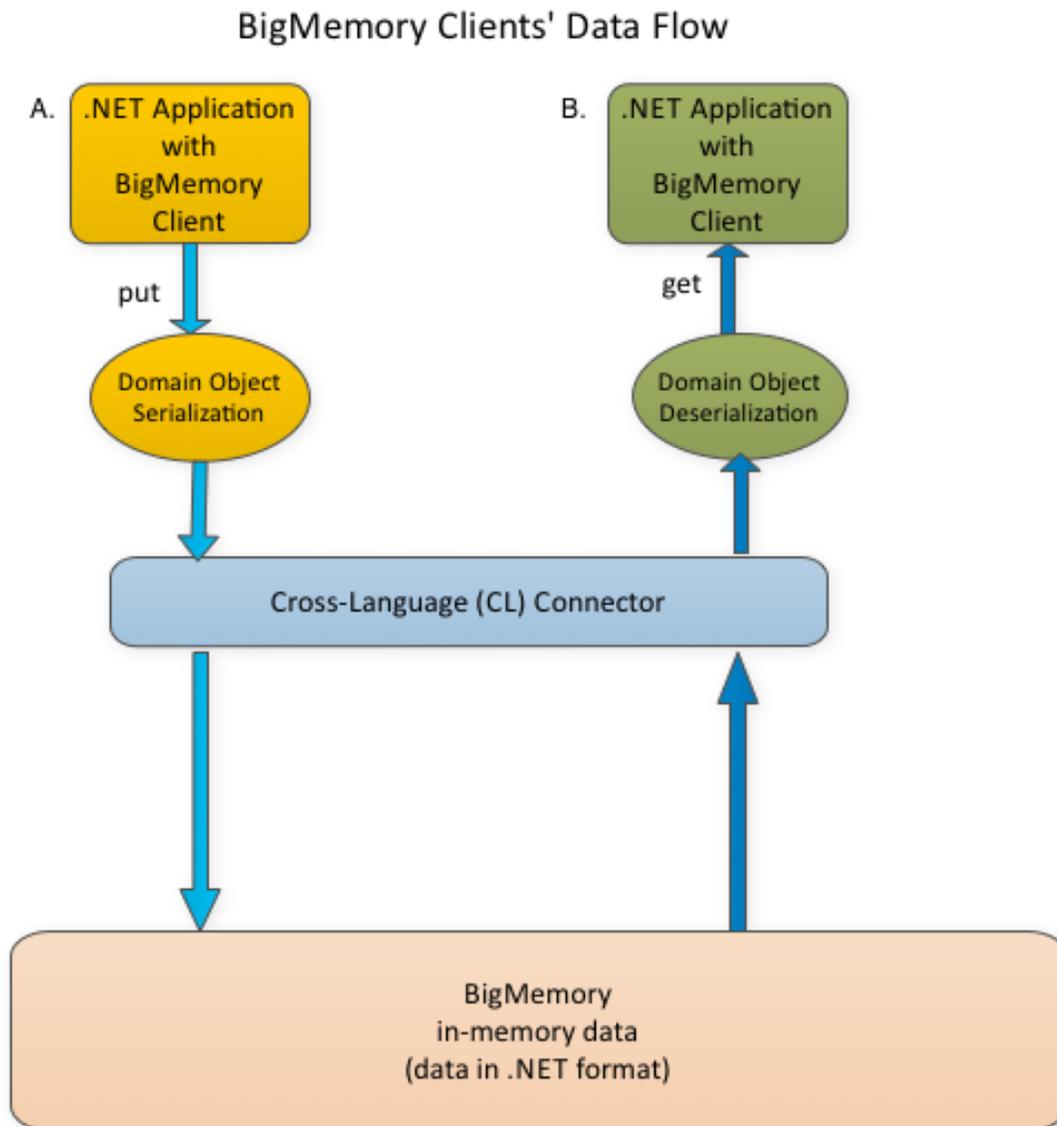
1. Data from a .NET client application can be stored in BigMemory using the BigMemory Client. A put from a BigMemory Client is serialized and then transported to the Cross-Language Connector and put into BigMemory.

2. Data can be retrieved from BigMemory by the same client or by a different client. A get from the client retrieves the serialized object from BigMemory and deserializes it, making the Domain Object available to the client.

Components

Cross-Language Connector - The CL Connector is configured with the cross-lang-config.xml file, and it accesses BigMemory data that is contained in caches that have been defined in the ehcache.xml file. Each CL Connector needs its own CacheManager, and if you want more than one CacheManager, you will need as many CL Connectors.

BigMemory Client for each platform (.NET or C++) - The BigMemory Client is specific to a platform (for example, C++, C#, etc.). It will convert data from the platform into platform-neutral format and send it to the CL Connector.

Deployment

BigMemory Cross-Language support is provided as a client-server model. One server potentially connects to a Terracotta Server Array (TSA) but could also be a standalone "caching server." Typically, in order to minimize latency, the server would run on the same hardware as the client(s). You can have one or more clients' processes then connecting to that server.

The two main CL Connector deployments are:

- As a standalone server process that runs in its own dedicated Java Virtual Machine. A standalone CL Connector is started and stopped using the batch or shell scripts provided in the kit.

- As an embedded server process within the CacheManager used by an existing Ehcache application. An embedded CL Connector is configured by making simple changes to the ehcache.xml file, and then it is started and stopped via your application.

Platform-neutral Format and Serialization

The platform-neutral format (PNF) is a Thrift-based, opaque, binary format that is used for wire transport between platforms. PNF objects can be trivially stored by BigMemory and the Terracotta Server Array. PNF data has the following characteristics:

- Consists of well-defined primitive types (string, integer, float, date, byte[], etc.)

- Keys are mapped to some type of primitive

- Values are comprised of two parts:

  - Binary byte array, opaque to Ehcache

  - Map of String::`<primitive type>` used for indexing/search

Data that will be shared across languages needs to be serialized. Cross-Language CacheSerializers are implementations which take a platform domain object and transform it into a platform-neutral format, and also perform the operation in reverse, from PNF back to the original format. In a mixed platform deployment, CacheSerializers need to be written for each platform, in the platform language.

The choice of which serializer to use depends upon your environment. Some planning and consideration needs to go into the choice of a serializer, as there are pros and cons

related to each. For more information, refer to "Cross-Language Serialization" on page 24.

In single-language environments, PNF is not necessary, and native serialization is used to interact with the CL Connector. For data that will be stored in its native format, you can get started without writing any serialization code by using the provided native serializer class. For more information, refer to "Single-Language Serialization" on page 26.

Installing and Configuring the .NET Client

This section includes the basic steps for setting up access to BigMemory from .NET applications.

Step 1: Verify you have the requirements

- .NET 3.5 or higher.

- Windows 2008 R2.

- JDK 1.6 or higher.

  > **Note:** With BigMemory Max 4.1.4 and higher, JDK 1.7 is required.

- An application server.

- Download and unpack the "BigMemory Max kit", which includes Cross-Language Support.

- Your license key (terracotta-license.key). Save the license key in your BigMemory home directory.

Step 2: Install the BigMemory Client

Install the BigMemory.NET Client by adding the following three .DLL files to your application:

- ${BIGMEMORY_HOME}/apis/csharp/bigmemory_csharp.dll

- ${BIGMEMORY_HOME}/apis/csharp/Nirvana DotNet.dll

- ${BIGMEMORY_HOME}/apis/csharp/Thrift.dll

Step 3: Install the Cross-Language Connector

Create the JVM that will host the BigMemory instance and the CL Connector, and add the following JAR files to the JVM's classpath.

- ${BIGMEMORY_HOME}/apis/ehcache/lib/ehcache-ee-<version>.jar

- ${BIGMEMORY_HOME}/apis/ehcache/lib/slf4j-api-<version>.jar

- ${BIGMEMORY_HOME}/apis/toolkit/lib/terracotta-toolkit-runtime-ee-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/commons-codec-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/commons-lang-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/commons-logging-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/httpclient-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/httpcore-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/libthrift-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/nServer-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/security-core-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/security-keychain-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/server-embedded-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/server-main-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/server-standalone-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/slf4j-jdk14-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/thrift-java-<version>.jar

Note: In the file names above, *<version>* is the current version of the JAR.

Step 4: Customize the cross-lang-config.xml file

A sample cross-lang-config.xml file is provided in the config-samples/ directory of the kit. You will need to customize it with the IP address and port where the CL Connector should bind. Below is an example:

```
<?xml version="1.0"?>
<xplatform
    xmlns="http://www.ehcache.org/xplatform"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ehcache.org/xplatform xplatform.xsd">
  <bind ip="*" port="8199" type="nirvana"/>
  <!--
  <workers min="10" max="100" />
  <serializer factoryClass="net.sf.ehcache.xplatform.serializer.MyFactory" />
  <secureinterface>
    <keystore location="/path/to/keystore.jks"/>
    <truststore location="/path/to/truststore.jks">
  </secureinterface>
  -->
</xplatform>
```

The optional configurations are:

- Worker threads that handle requests.

- Security information. For more information, refer to the *BigMemory Max Security Guide*.

- Serializer factory to be used to bind the CacheSerializer to the caches (necessary if your data is in more than one language format). MyFactory can implement the

default net.sf.ehcache.xplatform.serialization.CacheSerializerFactory. This class is responsible for creating CacheSerializers for each Cache of the CacheManager. For more information, refer to ".NET Client Serialization" on page 24.

> **Note:** If you will be working with .NET data only, comment out or remove the serializer line from the cross-lang-config.xml.

Step 5: Customize the ehcache.xml file

A sample ehcache.xml configuration file is provided in the config-samples/ directory of the kit. If you will be using the Terracotta Server Array, you will also need to customize the tc-config.xml file. For more information about these configuration files, refer to the *BigMemory Max Configuration Guide*.

If you will be deploying the CL Connector as an embedded server process in an existing CacheManager, add the EmbeddedXplatformServer class to your ehcache.xml file, with the `cfgFile` property pointing to the cross-lang-config.xml:

```
<ehcache name="existingCacheManager">
<cacheManagerEventListenerFactory
    class="net.sf.ehcache.xplatform.EmbeddedXplatformServer"
    properties="cfgFile=path/to/cross-lang-config.xml"/>
<!-- my caches and other ehcache config -->
</ehcache>
```

All the caches of that "existingCacheManager" will now be accessible from all BigMemory Clients. The server's lifecycle will be tied to the CacheManager. It will start as soon as you instantiate the CacheManager using the configuration, and it will shut down when you use net.sf.ehcache.CacheManager#shutdown.

Step 6: Start the CL Connector

This step is for starting the CL Connector as a standalone server process. If you have deployed the CL Connector as an embedded server process, it will be started and stopped through your application.

> **Note:** You might want to run the CL Connector as a Windows service. If so, see "Configuring the Terracotta Server to Run as a Service" in the *BigMemory Max Administrator Guide*.

In a terminal, change to your Terracotta server/ directory. Then run the start script with your cross-lang-config.xml and the ehcache.xml files:

```
%> cd /path/to/bigmemory-max-<version>/server
%> ./bin/start-cross-lang-connector.sh ../config-samples/cross-lang-config.xml
                                        ../config-samples/ehcache.xml
```

> **Note:** For Microsoft Windows installations, use the BAT scripts, and where forward slashes ("/") are given in directory paths, substitute back slashes ("\").

Optionally, you can include the `-pid <pidFileLocation>` argument to provide a file for storing the process identifier (pid) of the started server process (for example, `-pid /var/tmp/pid`). This `pidFileLocation` can later be used to stop the server.

Logging

By default, logging goes to console and to a file called bigmem-connector-%u.log, located in the directory that contains the start script. You can override this by including -Dxplatform.log=/path/to/my.log with the start script. You can also re-enable console logging by including -Dxplatform.log.console with the start script.

Stopping the server (for future reference)

Use the stop-cross-lang-connector.sh script in server/bin/ directory of your kit. You can pass as an argument either <pid> (the pid of the server to stop) or -pid <pidFileLocation> (the file containing the pid of the server to stop).

Accessing BigMemory Data from Your Application

This section provides basic code snippets for accessing BigMemory data from your application. For more explanation, refer to "Using the .NET Client API" on page 19. For the complete class library documentation, refer to the API documentation in the /apis/csharp/apidoc-csharp.zip directory in the kit.

Create a CacheManager

This snippet creates a CacheManager configuration that uses the Nirvana Shared Memory (SHM) transport and provisions a pool of 10 connections.

```
XPlatform.CreateCacheManager(new
Terracotta.Ehcache.Config.Configuration("/dev/shm", 10));
```

Retrieve a Cache from the CacheManager

This snippet can be used with the .NET client demo samples in the kit (see ".NET Client Demo Samples" on page 26). Notice that a serializer must be specified.

```
cacheManager.GetCache("TestCache", new RecipeSerializer());
```

Get or put entries into the cache

This snippet can be used with the .NET client demo samples in the kit (see ".NET Client Demo Samples" on page 26). Notice that you have the option to specify a consistency type for get and put operations.

```
cache.Put(recipe.name, recipe, ConsistencyType.STRONG);
```

Close the CacheManager

```
cacheManager.Close();
```

Search

Searches of BigMemory data can be made using the BigMemory Structured Query Language (SQL). Refer to "Search" on page 22.

Using the .NET Client API

This section covers connecting to, working with, and searching your BigMemory data. For the complete class library documentation, refer to the API documentation in the /apis/csharp/apidoc-csharp.zip directory of the kit.

Connecting with the CL Connector

Your application will communicate with the CL Connector via the ICacheManager interface.

Begin by creating a Configuration object. For a deployment where the BigMemory Client is located on the same machine as the CL Connector, use a configuration with the Nirvana Shared Memory (SHM) transport:

```
Configuration cfg=new Configuration("/dev/shm", 16));
```

For a deployment where the BigMemory Client and the CL Connector are located on different machines, use a configuration with the Nirvana socket transport. For example, here we create a configuration that connects to the local host at the default port, uses Nirvana socket transport, and provisions a pool of 16 connections to share:

```
Configuration cfg=new Configuration("localhost",
     Constants.defaultPort, TransportType.NIRVANA, 16);
```

Notice that configurations with different transport types require different parameters:

■ **Nirvana Shared Memory (SHM)** - `Configuration("SHMLocation", clientPoolSize)` - generally used when the BigMemory Client is located on the same machine as the CL Connector.

■ **Nirvana socket** - `Configuration("hostName", portNumber, TransportType.NIRVANA, clientPoolSize)` - used when the CL Connector and the BigMemory Client are located on different machines.

Once you have your Configuration instance ready, create an ICacheManager using the Xplatform.CreateCacheManager static method:

```
ICacheManager cachemanager = XPlatform.CreateCacheManager(cfg);
```

Shutting the connection down

The ICacheManager needs to be closed in order for it to release all resources (like connections). Close it by invoking the Close() method on it:

```
cachemanager.Close();
```

Accessing a Cache

Once you've obtained a reference to your ICacheManager, it is now connected to your CL Connector and will let you access any Cache known by the CL Connector.

To retrieve a thread-safe instance of a Cache, you will need its name, the type of the key, the value, and a serializer for those types.

Serializers

The protocol recognizes the following types:

- bool: A boolean value (true or false), one byte

- byte: A signed byte

- i16: A 16-bit signed integer

- i32: A 32-bit signed integer

- i64: A 64-bit signed integer

- double: A 64-bit floating point number

- byte[]: An array of bytes

- string: Encoding agnostic text or binary string

If you want to send complex types to the CL Connector, you will have to provide a serializer that can serialize your complex key and value types for a Cache.

Say we have the following Class, which is our Value type:

```
public class User
{
    public long Id { get; set; }
    public string Login { get; set; }
    public string Email { get; set; }
}
```

We store these keyed to the login, as a string. We now need to provide a serializer like this:

```
public class UserCacheSerializer : ICacheSerializer<string,User>
 {
    public Value SerializeKey(string key)
    {
        Value v = new Value();
        v.StringValue = key;
        return v;
    }
    public string DeserializeKey(Value serializedKey)
    {
        return serializedKey.StringValue;
    }
    public StoredValue SerializeValue(User objectValue)
    {
        Dictionary<String, Value> nvpairs = new Dictionary<string, Value>();
        Value idval = ValueHelper.NewValue(objectValue.Id);
        nvpairs["id"] = idval;
        Value emailval = ValueHelper.NewValue(objectValue.Email);
        nvpairs["email"] = emailval;
        StoredValue sv = new StoredValue();
        sv.Value = ValueHelper.NewValue(Encoding.Unicode.GetBytes(objectValue.Login));
        sv.Nvpairs = nvpairs;
        return sv;
    }
    public User DeserializeValue(StoredValue serializedValue)
    {
        User user = new User();
        user.Login = Encoding.Unicode.GetString(serializedValue.Value.BinaryValue);
```

```
        user.Id = serializedValue.Nvpairs["id"].LongValue;
        user.Email = serializedValue.Nvpairs["email"].StringValue;
        return user;
    }
}
```

Note that both serialize and deserialize methods (whether for key or value) are kept symmetric.

StoredValue is composed of the actual Value (in this example, the byte array representation of the user's login). StoredValue can hold entire object graphs serialized with whatever serialization strategies fits your needs. It also holds an arbitrary amount of name/value pairs. The names are String, while the values are of type Value. You can use these name-value pairs to store whatever you want. Note that these name-value pairs also become indexable and searchable using the search API.

A byte array is not indexable, however it can be used within name-value pairs. In the above example, both user id and email are stored as byte arrays. Even though byte arrays are not indexable, search can be enabled on the attributes. Since we use the login as the key in our use case, storing that as a binary representation within the "default unindexed value" is good enough.

Special handling for DateTime objects

C# captures date information with more precision than Java does, which means the string representation has parts that Java is not prepared to parse. There are two parts to handling this. First, use the IsoDateTimeConverter for the C# side serialization and deserialization code. Second, trim C# DateTime objects to millisecond granularity. You can do this via the Terracotta.Ehcache.Utilities.DateUtility.TrimToMillisGranularity(DateTime dt) method. It returns a new DateTime object trimmed appropriately. Alternatively, you can use ValueHelper.newValue(..) on a DateTime object to do the trimming for you.

Key-based store and retrieve

Now that we have defined an ICacheSerializer for our cache, we can retrieve the cache from the ICacheManager. The example below assumes that we have defined a cache named "userCache" within the ehcache.xml file used to configure the CL Connector.

```
// create a serializer
ICacheSerializer<string, User> serializer = new UserCacheSerializer();
// get a cache
ICache<string, User> users = cachemanager.GetCache("userCache", serializer);
// make a user
User user1 = new User();
user1.Id = 1;
user1.Email = "someone@somewhere";
user1.Login = "secretpasswd";
// put the user
users.Put(user1.Login, user1, ConsistencyType.STRONG);
```

The following is an example if you were using the BSON code sample for .NET (Recipe2 is the class defining the domain object, and Recipe2BsonSerializer() is the serializer class).

```
ICache<string, Recipe2> cache = cacheManager.GetCache("TestCache",
    new Recipe2BsonSerializer());
cache.Put(rec.Name, rec, ConsistencyType.STRONG);
```

The `ConsistencyType` attribute sets the consistency, strong or eventual, with which you expect the CL Connector to execute an operation. For more information, refer to "Consistency" on page 44.

To retrieve by key, invoke the get method:

```
User someUser = userCache.Get(someLogin, ConsistencyType.EVENTUAL);
```

To retrieve in C#:

```
Recipe2 rec = (Recipe2)cache.Get(args[1], ConsistencyType.EVENTUAL);
```

Compare and Swap (CAS) Operations

The following CAS operations are provided in C#:

■ PutIfAbsent(TKey, TValue) - Puts the specified key/value pair into the cache only if the key has no currently assigned value. Unlike Put, which can replace an existing key/value pair, PutIfAbsent creates the key/value pair only if it is not present in the cache.

■ Remove(TKey, TValue) - Conditionally removes the specified key, if it is mapped to the specified value.

■ Replace(TKey, TValue) - Maps the specified key to the specified value, if the key is currently mapped to some value.

■ Replace(TKey, OldValue, NewValue) - Conditionally replaces the specified key's old value with the new value, if the currently existing value matches the old value.

For more information about atomic operations and cache consistency, refer to "Consistency" on page 44.

Search

This section provides an overview of how to do cross-language searches of PNF data stored within BigMemory. More detailed information may be found on the *BigMemory Max Developer Guide*.

In order to search BigMemory data, you first need to add the `<searchable/>` tag to the cache configuration in your ehcache.xml file.

```
<ehcache>
(...)
  <cache name="userCache">
  (...)
    <searchable/>
  </cache>
</ehcache>
```

This configuration will scan all keys and values in the cache and, if they are of supported search types, add them as search attributes. Values of the Name/Value pairs in the PNF's StoredValue sent to the CL Connector, other than of type byte[], can all be indexed and made searchable. (Continuing with the example of the previous section, this would mean `id` and `email`.)

You can also add search attributes using the provided attributeExtractor class, for example:

```
<ehcache>
(...)
  <cache name="userCache">
  (...)
    <searchable>
      <searchAttribute name="email" class="net.sf.ehcache.xplatform.search.Indexer"/>
    </searchable>
  </cache>
</ehcache>
```

To perform a query, use the Cache.query(String) method, providing a BigMemory SQL query string, for example:

```
SearchResults<string, User> result = userCache.Query("select * from userCache
    where email ilike '%@terracottatech.com'");
```

For more information about how to construct BigMemory SQL queries, see "Searching with BigMemory SQL" in the *BigMemory Max Developer Guide*.

Note:	Note: Java (POJO) data stored within BigMemory can also be searched using the native Ehcache Search API, which is documented on the *BigMemory Max Developer Guide*.

Consistency

Consistency is set in the ehcache.xml configuration file, and it has to do with how the cache you connect to is clustered on the CL Connector. A cache clustered using strong consistency will only be able to have strong operations performed on it, meaning that changes made by any node in the cluster are visible instantly. Atomic operations (for example, putIfAbsent) are always performed in the strong mode. A cache clustered using eventual consistency, on the other hand, will be able to perform both strong and eventual operations, and changes made by any node are visible to other nodes eventually. To understand consistency at the Terracotta clustering level, refer to the *BigMemory Max Configuration Guide*.

In the Cross-Language API, the ConsistencyType attribute sets the consistency, strong or eventual, with which you expect the CL Connector to execute an operation. All operations on a cache that can be performed using either strong or eventual consistency take ConsistencyType as an additional parameter. Other than a few operations such as atomic ops, RemoveAll, and queries, most regular cache operations accept this last parameter.

When set to STRONG, data in the cache remains consistent across the cluster at all times. This mode guarantees that a read gets an updated value only after all write operations to that value are completed, and it guarantees that each put operation is in a separate transaction. When set to EVENTUAL, reads without locks are allowed, and read/write performance is substantially boosted, but at the cost of potentially having an inconsistent cache for brief periods of time.

If you try to execute an eventually consistent operation (for example, cache.get(key, EVENTUAL)) on a strong cache, you will get an exception. Executing a strongly consistent operations on an eventual cache is allowed but will result in locks being used on the

CL Connector, and your method will return only when all of the steps are finished (acquiring the appropriate lock, performing the operation, and releasing the lock).

.NET Client Serialization

Cross-Language Serialization

Data that will be shared across languages needs to be serialized. The choice of which serializer to use depends upon your environment. Some planning and consideration needs to go into the choice of a serializer, as there are pros and cons related to each. For example, with a BSON serializer, you can manually annotate the domain object for desired attributes. On the other hand, a Google Protocol Buffer serializer can automatically generate the domain object with the annotation for the serializer for you, however, if you have an existing domain object, then using the automatic generation can be intrusive.

Your serializer/deserializer class should include the following:

- A list of all the namespaces/packages that will be used.

- A class definition consisting of the following methods/functions:

    - serializeKey() - accepts the key and returns the platform-neutral format (PNF) Value

    - deserializeKey() - accepts the PNF Value and returns the key

    - serializeValue() - accepts the object value and returns the PNF StoredValue

    - deserializeValue() - accepts the PNF StoredValue and returns the object value

Example BSON serializer

```
using Newtonsoft.Json;
using Newtonsoft.Json.Bson;
using Newtonsoft.Json.Converters;
using Newtonsoft.Json.Linq;
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using Terracotta.Ehcache;
using Terracotta.Ehcache.Thrift;
```

Class definition: serializeKey()

```
public Terracotta.Ehcache.Thrift.Value serializeKey(string key)
    {
        return ValueHelper.newValue(key);
    }
```

Class definition: deserializeKey()

```
public string deserializeKey(Terracotta.Ehcache. Thrift.Value serializedKey)
 {
  return (string)ValueHelper.findSetValue  (serializedKey);
 }
```

Class definition: serializeValue()

```
public StoredValue serializeValue(Recipe2 objectValue)
 {
  MemoryStream ms = new MemoryStream();
// serialize recipe to BSON
  BsonWriter writer = new BsonWriter(ms);
  JsonSerializer serializer = new JsonSerializer();
  serializer.Serialize(writer, objectValue);
  byte[] barray = ms.ToArray();
  Dictionary<String, Object> nvPairs = new  Dictionary<String, Object>();
// two indexable attributes
  nvPairs["name"] = objectValue.Name;
  nvPairs["bakingTime"] = objectValue.BakingTime;
// ship the bson version of the recipe as a byte array
    return ValueHelper.nullSafeStoredValue(barray, nvPairs);
}
```

Class definition: deserializeValue()

```
public Recipe2 deserializeValue  (Terracotta.Ehcache.Thrift.StoredValue value)
 {
  MemoryStream ms = new MemoryStream(value.Value. BinaryValue);
  ms.Seek(0, SeekOrigin.Begin);
  BsonReader reader = new BsonReader(ms);
  JsonSerializer serializer = new JsonSerializer();
  Recipe2 recipe = serializer.Deserialize <Recipe2>(reader);
  return recipe;
 }
```

Example Google Protocol Buffer Serializer

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using Terracotta.Ehcache;
using Terracotta.Ehcache.Thrift;
```

Class definition: serializeKey()

```
public Value serializeKey(string key)
    {
            Value val = new Value();
             val.StringValue = key;
             return val;
    }
```

Class definition: deserializeKey()

```
public string deserializeKey(Value serializedKey)
    {
        return serializedKey.StringValue;
    }
```

Class definition: serializeValue()

```
public StoredValue serializeValue(RecipeProject.RecipeStructure objectValue)
 {
   Dictionary<String, Object> nvPairs = new
   Dictionary<String, Object>();
   nvPairs.Add(KEY_NAME, objectValue.name);
   MemoryStream ms = new MemoryStream();
   ProtoBuf.Serializer.Serialize<RecipeProject.
```

```
    RecipeStructure>(ms,objectValue);
    return ValueHelper.nullSafeStoredValue(ms.ToArray(), nvPairs);
}
```

Class definition: deserializeValue()

```
public RecipeProject.RecipeStructure deserializeValue (StoredValue value)
{
  MemoryStream ms = new MemoryStream(value.Value.BinaryValue);
  return ProtoBuf.Serializer.Deserialize<RecipeProject. RecipeStructure>(ms);
}
```

Single-Language Serialization

For single-language use cases, it is not necessary to create a serializer/deserializer class. Leverage .NET's native binary serialization by including the provided CSharpNativeSerializer class in cache operations. For example:

```
ICachemanager manager = XPlatform.CreateCacheManager(...);
ICache cache<string,string> = manage.GetCache("cachename",
    new CSharpNativeSerializer<string,string>())
```

API documentation for the CSharpNativeSerializer is available in the /apis/csharp/apidoc/ directory of the kit. For more information about .NET native serialization, refer to "http://msdn.microsoft.com/en-us/library/7ay27kt9%28v=vs.90%29.aspx".

.NET Client Demo Samples

The following examples are provided in the code-samples/ directory in the kit.

BSON - C# Example

The BSON - C# example consists of three files:

■ Recipe2.cs is the C# implementation of the recipe domain class. It includes classes for the Recipe2 class itself, plus support classes for things like Ingredient, MethodItem, etc., which are used by the Recipe2 class. In this class, some of the domain attributes have been annotated for the BSON serializer. It is not necessary to annotate in the domain. However, if you do not, then you will have to add some extra code to your serializer class.

■ Recipe2BsonSerializer.cs implements the CacheSerializer<string, Recipe2> interface. It is responsible for serializing and deserializing keys and values, moving between the Recipe2 domain object and the Thrift types Value and StoredValue. Serialization and deserialization of the Recipe2 objects is handled through the Json.net automatic BSON serializer.

■ Recipe2ConsoleApp.cs contains a main class, and tiny repository of pre-generated recipes. The console app understands get and put. Assuming you start a CL Connector on the same machine, you may do:

  ■ Recipe2ConsoleApp put "Mac&Cheese"

  ■ Recipe2ConsoleApp put "BreadedChicken"

- Recipe2ConsoleApp get "Mac&Cheese"

- Recipe2ConsoleApp get "BreadedChicken" where ("Mac&Cheese" and "BreadedChicken" are the two preloaded recipes).

To run the BSON - C# example:

1. Create a new Visual Studio project and add the three .CS recipe files.

2. Add the following libraries in the References section of your project:

   - bigmemory_csharp.dll (included in the kit)

   - Thrift.dll (included in the kit)

   - Nirvana DotNet.dll (included in the kit)

   - Newtonsoft.json.dll (available from http://json.codeplex.com/)

3. Start the server as described in "Installing and Configuring the .NET Client " on page 15, in the step "Start the CL Connector", and then run the console application (Recipe2ConsoleApp.cs).

Protocol Buffer - C# Example

The Protocol Buffer - C# example consists of the following files:

- recipe.cs is the C# implementation of the recipe domain class. This was automatically generated and has the annotation included in the domain.

- RecipeSerializer.cs implements the CacpeSerializer interface. Serialization and deserialization of the recipe object is handled by the Google Protocol Buffer serializer.

- SimpleRecipeConsoleApp.cs contains the main class and does simple get and put.

The example is a recipe package that includes ingredients, measurements, and instructions.

```
package RecipeProject;
message RecipeStructure {
            required string name = 1;
            required string description = 2;
            required int32 preparation_time = 3;
            required Method method = 6;
            required Ingredients ingredients = 5;
            optional int32 baking_time = 4;
            optional ServingInstructions instructions = 7;
}
message Ingredients {
            repeated Ingredient ingredient = 1;
}
message Ingredient {
            required float quantity = 1;
            required Measurement measurement = 2;
            required IngredientDescription ingredient = 3;
}
message Measurement {
            required string measurement_type = 1;
}
```

```
message IngredientDescription {
            required string ingredient = 1;
            optional Ingredient alternative_ingredient = 2;
}
message Method {
            repeated MethodItem items = 1;
}
message MethodItem {
            required int32 step_number = 1;
            required string instruction = 2;
}
message ServingInstructions {
            required int32 servings = 1;
            required Measurement serving_measurement = 2;
            optional Method method = 3;
}
```

To run the Protocol Buffer - C# example:

1. Create a new Visual Studio project and add the three .CS recipe files.

2. Add the following libraries in the References section of your project:

    ■ bigmemory_csharp.dll (included in the kit)

    ■ Thrift.dll (included in the kit)

    ■ Nirvana DotNet.dll (included in the kit)

    ■ protobuf-net.dll (available from http://code.google.com/p/protobuf-net/)

3. Start the server as described in "Installing and Configuring the .NET Client " on page 15, in the step "Start the CL Connector", and then run the console application (SimpleRecipeConsoleApp.cs).

| **Note:** | If you have already installed the BigMemory Client and Cross-Language Connector, you can run this example by adding the external protobuf dependency. The maven link is |

```
<dependency>
        <groupId>com.google.protobuf</groupId>
        <artifactId>protobuf-java</artifactId>
        <version>2.5.0</version>
</dependency>
```

This dependency will need to be in both BigMemory .Net Client and CL Connector classpaths. For the CL Connector, the component that needs to be set is EhcacheThriftConfiguration.Builder().cacheSerializerFactory(RecipeSerializerFactory.class) so that the RecipeSerializerFactory, RecipeSerializer, and Recipe classes will be in the CL Connector's classpath. For the BigMemory .Net Client, the component that will be run is SimpleRecipeConsoleApp, which requires the SimpleRecipeConsoleApp, RecipeSerializer, and Recipe classes to be on the classpath along with the BigMemory .Net Client dependencies.

3 BigMemory C++ Client

Concepts and Architecture

Concepts and Architecture

The BigMemory C++ Client and Cross-Language Connector enable interoperability between your C++ application and other platforms by providing access to BigMemory in-memory data.

BigMemory Cross-Language Clients supply access to data from multiple platforms. Supported platforms can interoperate using the same data. For example, a Java application can read data added by a C++ or .NET client, and vice versa. The BigMemory Clients also allow searching of the BigMemory data. There is no additional piece of infrastructure required.

A Java Virtual Machine (JVM) is set up on the same hardware that hosts the other platform's code. This JVM will host the Cross-Language Connector and the BigMemory instance. The Cross-Language Connector runs as a JVM process that talks to the BigMemory instance and opens the communication channel between other processes and BigMemory. The Cross-Language Connector runs as any other Terracotta client (L1) from the cluster perspective, but it allows other processes to connect to it using a well-defined protocol and API.

The BigMemory C++ Client connects and accesses data stored in BigMemory. The BigMemory Client is a library which is installed on the target platform in the appropriate path. Within the target application, the API is used to talk to the JVM.

Figure 1 below illustrates a topology where multiple clients share one Cross-Language Connector. The BigMemory Client is added to the application, and then the Cross-Language Connector provides access to the BigMemory in-memory data.

Figure 1. Cross-Language Topology, Example 1

Figure 2 below illustrates a topology with multiple Cross-Language Connectors. Hosting the CL Connector on the same machine as your application may have the advantage of keeping your data closer and reducing latency. Each CL Connector requires its own CacheManager, so this topology may be a good approach for keeping discrete data sets available to different applications.

Figure 2. Cross-Language Topology, Example 2

Factors to consider when designing your topology include: your data access patterns, how data is shared among your applications, the amount of memory you have, and the amount of data that you want keep in memory.

Figure 3 below illustrates the flow of data through the Cross-Language Connector.

Figure 3. BigMemory Cross-Language Clients' Data Flow

BigMemory Clients' Data Flow

1. Data from a C++ or .NET client application can be stored in BigMemory using the BigMemory Client. A put from a BigMemory Client is converted into platform-neutral format (PNF). It is then transported to the Cross-Language Connector and put into BigMemory, either directly or after serialization, depending upon the use case.

2. Data can be retrieved from BigMemory by the same client or by a different client. A get from the client retrieves the serialized object from BigMemory and deserializes it, making the Domain Object available to the client.

3. You may also have a Java client, which can store and access BigMemory data in either of these ways:

a. If the BigMemory data is primarily in platform-neutral format (PNF), use the Ehcache API through a CacheSerializer that lazily serializes/deserializes the data for use with your Java application.

b. If the BigMemory data is primarily in Java format, use the Ehcache API directly from Java. (F) In this case, the Cross-Language Connector for C++ or .NET clients should include a Java CacheSerializer.

Components

Cross-Language Connector - The CL Connector is configured with the cross-lang-config.xml file, and it accesses BigMemory data that is contained in caches that have been defined in the ehcache.xml file. Each CL Connector needs its own CacheManager, so if you want more than one CacheManager, you will need as many CL Connectors.

BigMemory Client for each platform - The BigMemory Client is specific to a platform (for example, C++, .NET, etc.). It will convert data from the platform into platform-neutral format and send it to the CL Connector.

Deployment

BigMemory Cross-Language support is provided as a client-server model. One server potentially connects to a Terracotta Server Array (TSA) but could also be a standalone "caching server." Typically, in order to minimize latency, the server would run on the same hardware as the client(s). You can have one or more clients' processes then connecting to that server.

The two main CL Connector deployments are:

■ As a standalone server process that runs in its own dedicated JVM. A standalone CL Connector is started and stopped using the batch or shell scripts provided in the kit.

■ As an embedded server process within the CacheManager used by an existing Ehcache application. An embedded CL Connector is configured by making simple changes to the ehcache.xml file, and then it is started and stopped via your application.

Platform-neutral Format and Serialization

The platform-neutral format (PNF) is a Thrift-based, opaque, binary format that is used for wire transport between platforms. PNF objects can be trivially stored by BigMemory and the Terracotta Server Array. PNF data has the following characteristics:

■ Consists of well-defined primitive types (string, integer, float, date, byte[], etc.)

■ Keys are mapped to some type of primitive

■ Values are comprised of two parts:

■ Binary byte array, opaque to Ehcache

■ Map of String::<primitive type> used for indexing/search

Data that will be shared across languages needs to be serialized. Cross-Language CacheSerializers are implementations which take a platform domain object and transform it into a platform-neutral format, and also perform the operation in reverse, from PNF back to the original format. In a mixed platform deployment, CacheSerializers need to be written for each platform, in the platform language.

For more information, see "C++ Client Serialization" on page 43.

Installing and Configuring the BigMemory C++ Client

This section includes the basic steps for setting up access to BigMemory from C++ applications.

1: Verify you have the requirements

- Microsoft Windows Server 2008 R2 (64-bit), or Red Hat Enterprise Linux 6.4 (32-bit or 64-bit)

- Microsoft Visual Studio 2012 VC110

- GNU G++ Compiler

- Oracle Java Development Kit 1.6 or higher (Note: With BigMemory Max 4.1.4 and higher, JDK 1.7 is required.)

- An application server

- Downloadand unpack the BigMemory Max kit, which includes Cross-Language Support.

- Your license key (terracotta-license.key). Save the license key in your BigMemory home directory.

2: Install the BigMemory Client

Install the BigMemory C++ Client by compiling your application with A and B below:

- A. Include ${BIGMEMORY_HOME}/apis/cpp/include/bigmemory

- B. Link to one of the following, depending upon whether you are using Linux or Windows:

  - ${BIGMEMORY_HOME}/apis/cpp/linux/x86

  - ${BIGMEMORY_HOME}/apis/cpp/linux/x86_64

  - ${BIGMEMORY_HOME}/apis/cpp/windows/win32

  - ${BIGMEMORY_HOME}/apis/cpp/windows/x64

3: Install the Cross-Language Connector

Create the JVM that will host the BigMemory instance and the CL Connector, and add the following JAR files to the JVM's classpath.

- ${BIGMEMORY_HOME}/apis/ehcache/lib/ehcache-ee-<version>.jar

- ${BIGMEMORY_HOME}/apis/ehcache/lib/slf4j-api-<version>.jar

- ${BIGMEMORY_HOME}/apis/toolkit/lib/terracotta-toolkit-runtime-ee-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/commons-codec-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/commons-lang-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/commons-logging-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/httpclient-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/httpcore-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/libthrift-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/nServer-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/security-core-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/security-keychain-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/server-embedded-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/server-main-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/server-standalone-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/slf4j-jdk14-<version>.jar

- ${BIGMEMORY_HOME}/server/lib/thrift-java-<version>.jar

Note: In the file names above, *<version>* is the current version of the JAR.

4: Customize the **cross-lang-config.xml** file

A sample cross-lang-config.xml file is provided in the config-samples/ directory of the kit. You will need to customize it with the IP address and port where the CL Connector should bind. Below is an example:

```
<?xml version="1.0"?>
<xplatform
    xmlns="http://www.ehcache.org/xplatform"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ehcache.org/xplatform xplatform.xsd">
  <bind ip="*" port="8199" type="nirvana"/>
  <!--
  <workers min="10" max="100" />
  <serializer factoryClass="net.sf.ehcache.xplatform.serializer.MyFactory" />
  <secureinterface>
    <keystore location="/path/to/keystore.jks"/>
    <truststore location="/path/to/truststore.jks">
  </secureinterface>
  -->
</xplatform>
```

The optional configurations are:

- The number of worker threads to handle requests.

- Security information. For more about security, refer to the *BigMemory Max Security Guide*.

- Serializer factory to be used to bind the CacheSerializer to the caches. MyFactory can implement the default net.sf.ehcache.xplatform.serialization.CacheSerializerFactory. This class is responsible for creating CacheSerializers for each Cache of the CacheManager. For more information, refer to ".NET Client Serialization" on page 24.

5: Customize the `ehcache.xml` file

A sample ehcache.xml configuration file is provided in the config-samples/ directory of the kit. If you will be using the Terracotta Server Array, you will also need to customize the tc-config.xml file. For more information about these configuration files, refer to the *BigMemory Max Configuration Guide*.

If you will be deploying the CL Connector as an embedded server process in an existing CacheManager, add the EmbeddedXplatformServer class to your ehcache.xml file, with the `cfgFile` property pointing to the cross-lang-config.xml:

```
<ehcache name="existingCacheManager">
<cacheManagerEventListenerFactory
    class="net.sf.ehcache.xplatform.EmbeddedXplatformServer"
    properties="cfgFile=path/to/cross-lang-config.xml"/>
<!-- my caches and other ehcache config -->
</ehcache>
```

All the caches of that "existingCacheManager" will now be accessible from all BigMemory Clients. The server's lifecycle will be tied to the CacheManager. It will start as soon as you instantiate the CacheManager using the configuration, and it will shutdown when you use net.sf.ehcache.CacheManager#shutdown.

6: Start the CL Connector

This step is for starting the CL Connector as a standalone server process. If you have deployed the CL Connector as an embedded server process, it will be started and stopped via your application.

Note: You might want to run the CL Connector as a Windows service. If so, see "Configuring the Terracotta Server to Run as a Service" in the *BigMemory Max Administrator Guide*.

In a terminal, change to your Terracotta server/ directory. Then run the start script with your cross-lang-config.xml and the ehcache.xml files:

```
%> cd /path/to/bigmemory-max-<version>/server
%> ./bin/start-cross-lang-connector.sh ../config-samples/cross-lang-config.xml
    ../config-samples/ehcache.xml
```

Note: For Microsoft Windows installations, use the BAT scripts, and where forward slashes ("/") are given in directory paths, substitute back slashes ("\").

Optionally, you can include the `-pid <pidFileLocation>` argument to provide a file for storing the process identifier (pid) of the started server process (for example, `-pid /var/tmp/pid`). This pidFileLocation can later be used to stop the server.

Logging

By default, logging goes to console and to a file called bigmem-connector-%u.log, located in the directory that contains the start script. You can override this by including `-Dxplatform.log=/path/to/my.log` with the start script. You can also re-enable console logging by including `-Dxplatform.log.console` with the start script.

Stopping the server (for future reference)

Use the stop-cross-lang-connector.sh script in server/bin/ directory of your kit. You can pass as an argument either `<pid>` (the pid of the server to stop) or `-pid <pidFileLocation>` (the file containing the pid of the server to stop).

Accessing BigMemory Data from Your Application

This section provides basic code snippets for accessing BigMemory data from your application. For more explanation, refer to "Using the .NET Client API" on page 19. For the complete class library documentation, refer to the API documentation in the /apis/cpp/apidoc-cpp.zip directory of the kit.

Create a CacheManager

This snippet creates a CacheManager configuration that uses the Nirvana Shared Memory (SHM) transport and provisions a pool of 10 connections.

```
XPlatform::createCacheManager(terracotta::ehcache::config::Configuration("/dev/shm", 10));
```

Retrieve a Cache from the CacheManager

This snippet can be used with the .NET client demo samples in the kit (see "demo code samples" on page 26). Notice that a serializer must be specified.

```
cacheManager->getCache(cacheName, RecipeProtoBufSerializer());
```

Get or put entries into the cache

This snippet can be used with the .NET client demo samples in the kit (see "demo code samples" on page 26). Notice that you have the option to call the configured consistency type for get and put operations.

```
cache->put(key, recipeStructure, type);
```

Search

Searches of BigMemory data can be made using the BigMemory Structured Query Language (SQL). Refer to the *Search* section in "Using the C++ Client API" on page 39.

Using the C++ Client API

This section covers connecting to, working with, and searching your BigMemory data. For the complete class library documentation, refer to the API documentation in the /apis/csharp/apidoc-cpp.zip directory of the kit.

Connecting with the CL Connector

Your application will communicate with the CL Connector via the CacheManager interface.

Begin by creating a Configuration object. For a deployment where the BigMemory Client is located on the same machine as the CL Connector, use a configuration with the Nirvana Shared Memory (SHM) transport:

```
Configuration("/dev/shm", 16);
```

For a deployment where the BigMemory Client and the CL Connector are located on different machines, use a configuration with the Nirvana socket transport. For example, here we create a configuration that connects to the local host at port 8199, and provisions a pool of 16 connections to share:

```
Configuration("localhost", 8199, 16);
```

Notice that configurations with different transport types require different parameters:

- **Nirvana Shared Memory (SHM)** - `Configuration("SHMLocation", clientPoolSize)` - generally used when the BigMemory Client is located on the same machine as the CL Connector.

- **Nirvana socket** - `Configuration("hostName", portNumber, clientPoolSize)` - used when the CL Connector and the BigMemory Client are located on different machines.

Once you have your Configuration instance ready, create a CacheManager using the XPlatform static method:

```
CacheManager cacheManager = XPlatform::createCacheManager(config);
```

Shutting the connection down

The CacheManager needs to be closed in order for it to release all resources (like connections). Close it by invoking the destructor:

```
delete cacheManager;
```

Accessing a Cache

Once you've obtained a reference to your CacheManager, it is now connected to your CL Connector and will let you access any Cache known by the CL Connector.

To retrieve a thread-safe instance of a Cache, you will need its name, the type of the key, the value, and a serializer for those types.

Serializers

The protocol recognizes the following types:

- bool: A boolean value (true or false), one byte

- byte: A signed byte

- int: An 8-bit signed integer

- i16: A 16-bit signed integer

- i32: A 32-bit signed integer

- i64: A 64-bit signed integer

- double: A 64-bit floating point number

- byte[]: An array of bytes

- string: Encoding agnostic text or binary string

If you want to send complex types to the CL Connector, you will have to provide a serializer that can serialize your complex key and value types for a Cache.

Say we have the following Class, which is our Value type:

```
class User{
private: long Id;
    std::string Login;
    std::string Email;
public:  int getID() { return Id; }
                    void setID(int id) { Id = id; }
                    string getEmail() { return Email; }
                    void setEmail(int email) { Email = email; }
                    string getLogin() { return Login; }
                    void setLogin(string name) { Login = name; }
};
```

We store these keyed to the login, as a string. We now need to provide a serializer like this:

```
class UserCacheSerializer : public CacheSerializer<std::string, User>
 {
    ValueObject serializeKey(std::string &deserializedKey)
    {
        return ValueObject::stringValue(deserializedKey);
    }
    std::string deserializeKey(ValueObject &serializedKey)
    {
        return serializedKey.getString();
    }
    CacheValue serializeValue(User objectValue)
    {
        std::map<std::string, ValueObject> nvpairs;
        nvpairs["id"] = ValueObject::int8Value(objectValue.Id);
        nvpairs["email"] = ValueObject::stringValue(objectValue.Email);
        CacheValue value = CacheValue();
        value.Value = ValueObject::stringValue(objectValue.Login);
        value.setNvPairs(nvpairs);
        return value;
    }
    User deserializeValue(CacheValue serializedValue)
```

```
    {
        User user;
        user.Login = serializedValue.getValue().getString();
        user.Id = serializedValue.getNvPairs()["id"].getInt8();
        user.Email = serializedValue.serializedValue.getNvPairs()["email"].getString();
        return user;
    }
}
```

Note that both serialize and deserialize methods (whether for key or value) are kept symmetric.

CacheValue is composed of the actual ValueObject (in this example, the string representation of the user's login). CacheValue can hold entire object graphs serialized with whatever serialization strategies fits your needs. It also holds an arbitrary amount of name/value pairs. The names are std::string, while the values are of type ValueObject. You can use these name-value pairs to store whatever you want. Note that these name-value pairs also become indexable and searchable using the search API.

A byte array is not indexable, however it can be used within name-value pairs. In the above example, both user id and email are stored as byte arrays. Even though byte arrays are not indexable, search can be enabled on the attributes. Since we use the login as the key in our use case, storing that as a binary representation within the "default unindexed value" is good enough.

Note: The RawCache class, available directly from the CacheManager, allows you to access raw data, i.e., ValueObject as keys, and CacheValue as values. RawCache can be used when your data does not need to be serialized, and it allows you to create a cache without a serializer.

Key-based store and retrieve

Now that we have defined a CacheSerializer for our cache, we can retrieve the cache from the CacheManager. The example below assumes that we have defined a cache named "userCache" within the ehcache.xml file used to configure the CL Connector.

```
// create a serializer
CacheSerializer<string, User> *serializer = new userCacheSerializer();
// get a cache
Cache<string, User> users = cacheManager->getCache("userCache", *serializer);
// make a user
User user1;
user1.Id = 1;
user1.Email = "someone@somewhere";
user1.Login = "secretpasswd";
// put the user
users->put(user1.Login, user1, type);
```

The following is an example for the Recipe code sample (example03). Recipe is the class defining the domain object, and RecipeProtoBufSerializer() is the serializer class.

```
Cache<string, Recipe> *cache = cacheManager->getCache("TestCache",
    new RecipeProtoBufSerializer());
cache->put(rec.Name, rec, type);
```

The `type` attribute calls the consistency, configured as either strong or eventual, with which you expect the CL Connector to execute the operation. For more information, refer to "Consistency" on page 44.

To retrieve by key, invoke the get method:

```
User someUser = userCache->get(someLogin, type);
```

An example of retrieving, using the Recipe demo:

```
Recipe rec = (Recipe)cache->get(args[1], type);
```

Compare and Swap (CAS) Operations

The following CAS operations are provided in C++:

- putIfAbsent(key_type_ref key, value_type_ref cacheValue) - Puts the specified key/value pair into the cache only if the key has no currently assigned value. Unlike put, which can replace an existing key/value pair, putIfAbsent creates the key/value pair only if it is not present in the cache.

- remove(key_type_ref key, value_type_ref value) - Conditionally removes the specified key, if it is mapped to the specified value.

- replace(key_type_ref key, value_type_ref value) - Maps the specified key to the specified value, if the key is currently mapped to some value.

- replace(key_type_ref key, value_type_ref oldValue, value_type_ref newValue) - Conditionally replaces the specified key's old value with the new value, if the currently existing value matches the old value.

For more information about atomic operations and cache consistency, refer to "Consistency" on page 44.

Search

This section provides an overview of how to do cross-language searches of PNF data stored within BigMemory. More detailed information may be found in "Searching a Cache" in the *BigMemory Max Developer Guide*.

In order to search BigMemory data, you first need to add the `<searchable/>` tag to the cache configuration in your ehcache.xml file.

```
<ehcache>
(...)
  <cache name="userCache">
  (...)
    <searchable/>
  </cache>
</ehcache>
```

This configuration will scan all keys and values in the cache and, if they are of supported search types, add them as search attributes. Values of the Name/Value pairs in the PNF's ValueObject sent to the CL Connector, other than of type byte[], can all be indexed and made searchable. (Continuing with the example of the previous section, this would mean `id` and `email`.)

You can also add search attributes using the provided attributeExtractor class, for example:

```
<ehcache>
(...)
```

```
  <cache name="userCache">
  (...)
    <searchable>
      <searchAttribute name="email" class="net.sf.ehcache.xplatform.search.Indexer"/>
    </searchable>
  </cache>
</ehcache>
```

To perform a query, use the Cache.query(String) method, providing a BigMemory SQL query string, for example:

```
SearchResults<string, User> result = userCache->query("select * from userCache
    where email ilike '%@terracottatech.com'");
```

For more information about how to construct BigMemory SQL queries, see "Searching with BigMemory SQL" in the *BigMemory Max Developer Guide*.

Note: Java (POJO) data stored within BigMemory can also be searched using the native Ehcache Search API, which is documented in the *BigMemory Max Developer Guide*.

C++ Client Serialization

Data that will be shared across languages needs to be serialized. The choice of which serializer to use depends upon your environment. Some planning and consideration needs to go into the choice of a serializer, as there are pros and cons related to each. For example, with a JSON serializer, you can manually annotate the domain object for desired attributes. On the other hand, a Google Protocol Buffer serializer can automatically generate the domain object with the annotation for the serializer for you, however, if you have an existing domain object, then using the automatic generation can be intrusive.

Your serializer/deserializer class should include the following:

- A list of all the namespaces/packages that will be used.

- A class definition consisting of the following methods/functions:

  - serializeKey() - accepts the key and returns the platform-neutral format (PNF) Value

  - deserializeKey() - accepts the PNF Value and returns the key

  - serializeValue() - accepts the object value and returns the PNF ValueObject

  - deserializeValue() - accepts the PNF ValueObject and returns the object value

The following serializer/deserializer example demonstrates Google Protocol Buffer serialization:

```
#include "RecipeProtoBufSerializer.h"
ValueObject RecipeProtoBufSerializer::serializeKey(std::string &deserializedKey) {
    return ValueObject::stringValue(deserializedKey);
}
std::string RecipeProtoBufSerializer::deserializeKey(ValueObject &serializedKey) {
    return serializedKey.getString();
}
CacheValue RecipeProtoBufSerializer::serializeValue(RecipeStructure &deserializedValue) {
    CacheValue value = CacheValue();
```

```
    value.setValue(ValueObject::binaryValue(deserializedValue.SerializeAsString()));
    std::map<std::string, ValueObject> nvPairs;
    nvPairs[KEY_NAME] = ValueObject::stringValue(deserializedValue.name());
    value.setNvPairs(nvPairs);
    return value;
}
RecipeStructure RecipeProtoBufSerializer::deserializeValue(CacheValue &serializedValue) {
    RecipeStructure recipe = RecipeProject::RecipeStructure();
    if (serializedValue.isValueSet()) {
        recipe.ParseFromString(serializedValue.getValue().getBinary());
    }
    return recipe;
}
```

Consistency

Consistency is set in the ehcache.xml configuration file, and it has to do with how the cache you connect to is clustered on the CL Connector. A cache clustered using strong consistency will only be able to have strong operations performed on it, meaning that changes made by any node in the cluster are visible instantly. Atomic operations (for example, putIfAbsent) are always performed in the strong mode. A cache clustered using eventual consistency, on the other hand, will be able to perform both strong and eventual operations, and changes made by any node are visible to other nodes eventually. To understand consistency at the Terracotta clustering level, refer to the BigMemory documentation.

In the Cross-Language API, the `ConsistencyType` attribute sets the consistency, strong or eventual, with which you expect the CL Connector to execute an operation. For example:

```
Consistency::Type type = Consistency::STRONG;
```

All operations on a cache that can be performed using either strong or eventual consistency take `type` as an additional parameter. Other than a few operations such as atomic ops, RemoveAll, and queries, most regular cache operations accept this last parameter.

When set to STRONG, data in the cache remains consistent across the cluster at all times. This mode guarantees that a read gets an updated value only after all write operations to that value are completed, and it guarantees that each put operation is in a separate transaction. When set to EVENTUAL, reads without locks are allowed, and read/write performance is substantially boosted, but at the cost of potentially having an inconsistent cache for brief periods of time.

If you try to execute an eventually consistent operation on a strong cache, you will get an exception. Executing a strongly consistent operations on an eventual cache is allowed but will result in locks being used on the CL Connector, and your method will return only when all of the steps are finished (acquiring the appropriate lock, performing the operation, and releasing the lock).

C++ Client Demo Sample

The following sample is provided in the code-samples/cross-lang/example03/ directory of the kit. The demo is a recipe app that includes ingredients, measurements, and instructions. It uses Google Protocol Buffer serialization and consists of the following files:

■ main.cpp

■ recipe.pb.cc

■ recipe.pb.h

■ RecipeProtoBufSerializer.cpp

■ RecipeProtoBufSerializer.h

To compile and run the demo (Linux):

1. Install the Cross-Language Connector and customize the cross-lang-config.xml file, as described on the "Installing and Configuring the BigMemory C++ Client" on page 35 page.

2. Start the server as described on the "Installing and Configuring the BigMemory C++ Client" on page 35 page.

   Note: If you have already installed the BigMemory Client and Cross-Language Connector, you can run this example by adding the external protobuf dependency. The maven link is

   ```
   <dependency>
           <groupId>com.google.protobuf</groupId>
           <artifactId>protobuf-java</artifactId>
           <version>2.5.0</version>
   </dependency>
   ```

   This dependency will need to be in both BigMemory Client and CL Connector classpaths. For the CL Connector, the component that needs to be set is EhcacheThriftConfiguration.Builder().cacheSerializerFactory(RecipeSerializerFactory.class) so that the RecipeSerializerFactory, RecipeProtoBufSerializer, and recipe classes will be in the CL Connector's classpath. For the BigMemory Client, the component that will be run is main.cpp, which requires the main, recipe, and RecipeProtoBufSerializer classes to be on the classpath along with the BigMemory Client dependencies.

3. Run the application. Use G++ to compile and link to platform and serializer libraries. Set the output for console. For example:

   ```
   g++ -I<BMKIT>/apis/cpp/include -I<GoogleProtocolBufferInclude>
       -L<BMKIT>/apis/cpp/<PLATFORM>/<ARCH> -L<GoogleProtocolBufferLib>
       -lBigMemoryConnector -lprotobuf *.cc *.cpp -o console
   ```

4   **Security**

Cross-Language Connector Security

Note: For a brief overview of Terracotta security with links to individual topics, see the *BigMemory Max Security Guide*.

SSL security for BigMemory Cross-Language Clients requires setup for two main connections:

■ Between the Terracotta Server Array (TSA) and the Cross-Language (CL) Connector

■ Between the CL Connector and the BigMemory .NET or C++ client

Because the CL Connector is a client of the TSA, security setup for this connection is the same as the setup between an application server (a client of the Terracotta server) and its Terracotta server. (The app server or CL Connector is referred to as the L1, and the Terracotta server is referred to as the L2.) For information about the security between the TSA and the CL Connector, see the *BigMemory Max Security Guide*.

Security Between Connector and Client

Security between the CL Connector and the BigMemory Client

For setting up SSL security between the CL Connector and the BigMemory Client, ensure that the following configurations are in place:

1. The CL Connector's tc-config.xml must contain the necessary security references:

    ■ An L2 keystore with the certificate

    ■ An L2 keychain to open the keystore

    ■ An authentication file, with the user for the CL Connector

    For an example tc-config.xml, refer to the configuration example in the *BigMemory Max Security Guide*.

2. The TSA security files must be in the correct location.

    If your tc-config.xml file references relative paths, the security files must be located correctly. For example, if your path is `<url>file:keys/keyChain-relative.key</url>`, the files must be in the keys subdirectory under the Terracotta server installation.

3. The CL Connector username and a password must be the same as those stored in the TSA's auth file.

    The username is stored in the CL Connector's ehcache.xml file, for example:
    ```
    //non-secured:
    <terracottaConfig url="localhost:9510"/>
    //secured:
    <terracottaConfig url="admin@localhost:9510"/>
    ```

The password is stored in the CL Connector's keychain, and the keychain location can be given through a system property. For example:

```
-Dcom.tc.security.keychain.url=file:/path/to/CrossLanguage/keys/l1keychain.key
```

4. The cross-lang-config.xml file must indicate the truststore and keystore of the CL Connector:

```
<?xml version="1.0"?>
  <xplatform xmlns="http://www.ehcache.org/xplatform"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation=
          "http://www.ehcache.org/xplatform ../../main/xsd/xplatform.xsd">
    <bind ip="*" port="8199" type="nirvana"/>
    <secureinterface>
      <keystore location="/path/to/CL-keystore.jks"/>
      <truststore location="/path/to/CL-truststore.jks" password="123"/>
    </secureinterface>
  </xplatform>
```

The keystore contains the certificate for the security between the BigMemory Client and the CL Connector. The truststore contains the certificate of the TSA, that is, it holds the list of trusted parties you intend to communicate with.

5. Depending upon your security setup, you may need to start the CL Connector with some of the following system properties:

```
-Dcom.tc.security.keychain.url=file:/path/to/CrossLanguage/keys/l1keychain.key
-DSecretProvider.secret=secret
-Djavax.net.ssl.trustStore=keys/CL-truststore.jks
-Djavax.net.ssl.trustStorePassword=password
-Dtc.ssl.trustAllCerts=true
-Dtc.ssl.disableHostnameVerifier=true
```

Security for the BigMemory Client

Security for the BigMemory Client

1. Provide the client keychain with an entry for the keystore location, for example:

```
..\..\tools\security\bin\keychain.bat keys/l1keychain.key
         keys/CL-keystore.jks
```

2. Add self-signed certificates to the truststore.

   a. Add the client certificate.

   b. Add the truststore.

To add the client certificate for Windows:

1. Open the Start menu, click Run, and enter "certmgr.msc".

2. In the new window, expand the "Personal" folder and right click on the "Certificates" folder.

3. Select "All Tasks->Import...".

4. Follow the instructions and import the client certificate, SelfSignedCert.crt.

To add the truststore for Windows:

1. Open the Start menu, click Run, and enter "certmgr.msc".

2. In the new window, expand the "Trusted Root Certification Authorities" folder and right-click the "Certificates" folder.

3. Select "All Tasks->Import...".

4. Follow the instructions and import the truststore, SelfSignedCert.crt.