Notice

Copyright

Copyright © 2018 Pivotal Software, Inc. All rights reserved.

Pivotal Software, Inc. believes the information in this publication is accurate as of its publication date. The information is subject to change without notice. THE INFORMATION IN THIS PUBLICATION IS PROVIDED "AS IS." PIVOTAL SOFTWARE, INC. ("Pivotal") MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Use, copying, and distribution of any Pivotal software described in this publication requires an applicable software license.

All trademarks used herein are the property of Pivotal or their respective owners.
Contents

Preface: Contacting Pivotal Support ................................................................. x

Part II: Pivotal GemFire 8.2.x Native Client User's Guide ............................... 11

Part III: Pivotal GemFire Native Client 8.2 Release Notes .............................. 12
  What's New in Pivotal GemFire Native Client 8.2 ........................................... 13
  Installing Pivotal GemFire Native Client 8.2 .................................................. 14
  GemFire Native Client 8.2 Resolved Issues .................................................. 15
    Issues Resolved in Pivotal GemFire Native client 8.2.7 ......................... 15
    Issues Resolved in Pivotal GemFire Native client 8.2.6 ......................... 15
    Issues Resolved in Pivotal GemFire Native client 8.2.5 ......................... 15
    Issues Resolved in Pivotal GemFire Native client 8.2.4 ......................... 15
    Issues Resolved in Pivotal GemFire Native client 8.2.3 ......................... 15
    Issues Resolved in Pivotal GemFire Native client 8.2.2 ......................... 15
    Issues Resolved in Pivotal GemFire Native client 8.2.1 ......................... 15
    Issues Resolved in Pivotal GemFire Native client 8.2.0 ......................... 16

Part IV: Pivotal GemFire Native Client Supported Configurations .............. 17

Part V: Getting Started with Pivotal GemFire Native Client ........................... 21
  About the Native Client .................................................................................. 21
  Installing the Native Client .......................................................................... 22
    Installing on Linux and Solaris ................................................................. 23
    Installing on Windows ................................................................................ 24
    Uninstalling the Native Client ................................................................... 26
  Running Native Client Applications ............................................................. 26
    Developing C++ Programs on Linux .......................................................... 27
    Developing C++ Programs on Solaris ......................................................... 29
    Developing C++ Programs on Windows ..................................................... 30
  QuickStart Examples and Guide ................................................................. 31
    Running the QuickStart Examples ............................................................. 32
    Accessing the QuickStart Guide ............................................................... 33

Part VI: Setting System Properties ................................................................. 34
  Configuring the Native Client and Cache Server .......................................... 34
    Native Client Configuration ........................................................................ 35
    Cache Server Configuration ........................................................................ 36
    Attribute Definition Priority ...................................................................... 37
    Search Path for Multiple gfcpp.properties Files ...................................... 38
    Overriding gfcpp.properties Settings ....................................................... 39
    Defining Properties Programatically ........................................................ 40
    Attributes in gfcpp.properties .................................................................... 40
    gfcpp.properties Example File .................................................................... 45
    Using the Default Sample File .................................................................... 46
Part VII: Configuring the Native Client Cache

Caches................................................................................................................................................. 48
   About the Native Client Cache........................................................................................................ 48
   Cache APIs........................................................................................................................................... 49
   Local, Remote, and Distributed Caches............................................................................................ 50
   Creating and Accessing a Cache......................................................................................................... 51
   Closing the Cache................................................................................................................................. 52

Cache Initialization File (cache.xml).................................................................................................. 53
   Cache Initialization File Basics........................................................................................................... 54
   Example cache.xml File...................................................................................................................... 55
   Native Client Cache XML DTD............................................................................................................ 56

Regions.................................................................................................................................................... 59
   Declarative Region Creation................................................................................................................ 61
   Programmatic Region Creation............................................................................................................ 62
   Invalidating and Destroying Regions.................................................................................................. 63
   Region Access........................................................................................................................................ 64
   Getting the Region Size....................................................................................................................... 65

Region Entries.......................................................................................................................................... 65
   Entry Distribution Requirements........................................................................................................ 66
   Registering Interest for Entries............................................................................................................ 67
   Using serverKeys to Retrieve a Set of Region Keys............................................................................ 69
   Adding Entries to the Cache................................................................................................................ 70
   Updating Entries.................................................................................................................................... 71
   Accessing Entries.................................................................................................................................. 72
   Invalidating or Destroying Cached Entries......................................................................................... 73
   Notification for Operations................................................................................................................... 74

Region Consistency................................................................................................................................. 74

Region Attributes................................................................................................................................. 75
   Specifying Region Attributes.............................................................................................................. 76
   RegionShortcuts.................................................................................................................................... 77
   Mutable and Immutable Region Attributes.......................................................................................... 79
   CachingEnabled..................................................................................................................................... 80
   InitialCapacity....................................................................................................................................... 81
   LoadFactor............................................................................................................................................. 82
   ConcurrencyLevel................................................................................................................................. 83
   ConcurrencyChecksEnabled.................................................................................................................. 84
   LruEntriesLimit...................................................................................................................................... 85
   PersistenceManager.............................................................................................................................. 86
   Specifying Application Plug-In Attributes........................................................................................ 87
   Specifying Expiration Attributes........................................................................................................ 92

Cache Management............................................................................................................................... 98
   Client-to-Server Connection Process................................................................................................. 99
   Controlling Cache Size....................................................................................................................... 100
   Managing the Lifetime of a Cached Object......................................................................................... 101
   Using Thread Safety in Cache Management....................................................................................... 103
   Troubleshooting................................................................................................................................. 104

Part VIII: Pivotal GemFire Native Client C++ API.............................................................................. 105
   About the Native Client C++ API....................................................................................................... 105
   Creating a Cache................................................................................................................................. 107
   Creating a Proxy Client-Side Region................................................................................................. 107
   Adding an Entry to the Cache................................................................................................................ 108
**Part XII: Remote Querying** ................................................................. 205
- Remote Querying Basics .................................................................................. 205
  - Examples Data and Class Definitions .......................................................... 206
  - Executing a Query from the Native Client .................................................. 208
  - Querying the Portfolios Region ................................................................. 209
  - Modifying Cache Contents ......................................................................... 211
  - Creating Indexes ......................................................................................... 212
  - Remote Querying Requirements ............................................................... 213
- Using Query Strings in the Native Client ...................................................... 214
  - FROM Clause ............................................................................................... 215
    - Using Iterator Variables ........................................................................... 216
    - Importing and Using Object Classes ...................................................... 216
    - Predefined Class Types ......................................................................... 216
    - Specifying the Object Types of FROM Clause Collections .................. 217
  - SELECT Projection List ............................................................................... 219
  - SELECT Statement Query Results ............................................................ 220
  - WHERE Clause ............................................................................................ 222
  - Joins ........................................................................................................... 223
- Accessing Cached Data .................................................................................. 223
  - Basic Region Access .................................................................................. 224
  - Attribute Visibility ....................................................................................... 225
  - Modifying Query Scope .............................................................................. 226
  - Nested Query Scopes .................................................................................. 227
  - When Names Cannot Be Resolved ............................................................. 228
- Query Language Elements .......................................................................... 228
  - Method Invocation ....................................................................................... 229
  - Query Language Literals Supported by Native Client ............................... 230
  - Type Conversions ....................................................................................... 231
- Remote Query API ........................................................................................ 231
  - Creating and Managing Queries ............................................................... 232
  - Query Result Sets ....................................................................................... 233
  - Query Code Samples Returning ResultSet .............................................. 234
  - Query Code Samples Returning StructSet ............................................... 235

**Part XIII: Continuous Querying** ............................................................ 238
- How Continuous Querying Works ............................................................... 238
- Implementing a Continuous Query ............................................................. 240
  - Configuring Your System for Continuous Querying .................................. 242
  - Writing the Continuous Query ................................................................. 243
  - Writing the CQ Listener or CQ Status Listener ........................................ 244
  - CqEvent Object .......................................................................................... 249
  - Running the Continuous Query Code ....................................................... 250
  - CQ Execution Options ............................................................................... 252
  - When an Error Occurs in a Running CQ .................................................... 253
- Managing Continuous Queries ..................................................................... 253
- CQ API and Main Features .......................................................................... 254

**Part XIV: Using Connection Pools** ....................................................... 256
- How Client Load Balancing Works ............................................................. 256
- Server Locators ........................................................................................... 257
- Connection Pools ....................................................................................... 258
- Discovering Locators Dynamically ............................................................ 259
Contents

Pivotal GemFire Native Client

Operating System Statistics ............................................................................................................ 320
Linux Process Statistics .............................................................................................................. 321
Solaris Process Statistics ........................................................................................................... 322
Windows Process Statistics ....................................................................................................... 323

Part XXI: Installing the SQLite Persistence Manager ............................................................. 327
  Linux Installation ..................................................................................................................... 327
  Solaris Installation .................................................................................................................. 328
  Windows Installation .............................................................................................................. 329

Part XXII: Glossary ..................................................................................................................... 330
Contacting Pivotal Support

To contact Pivotal support, please submit a ticket from the Pivotal Support Page at http://support.pivotal.io/.
Part II

Pivotal GemFire 8.2.x Native Client User's Guide


Intended Audience

Pivotal GemFire Native Client User's Guide is intended for anyone who wants to use the Pivotal GemFire native client, including C++ and .NET developers programming their applications to use GemFire. This guide assumes experience developing with C++ and .NET. For information on how to manage and configure Pivotal GemFire deployments, see the Pivotal GemFire User's Guide.
Part III

Pivotal GemFire Native Client 8.2 Release Notes

This section contains release notes for all Pivotal GemFire Native Client 8.2.x releases.
What's New in Pivotal GemFire Native Client 8.2

This section lists new features in GemFire Native Client 8.2, with the most recent release shown first.

**Pivotal GemFire Native Client 8.2.7** is a maintenance release. See [Issues Resolved in Pivotal GemFire Native client 8.2.7](#).

**Pivotal GemFire Native Client 8.2.6** is a maintenance release. See [Issues Resolved in Pivotal GemFire Native client 8.2.6](#).

**Pivotal GemFire Native Client 8.2.5** is a maintenance release. See [Issues Resolved in Pivotal GemFire Native client 8.2.5](#).

**Pivotal GemFire Native Client 8.2.4** is a maintenance release. See [Issues Resolved in Pivotal GemFire Native client 8.2.4](#).

**Pivotal GemFire Native Client 8.2.3** is a maintenance release. See [Issues Resolved in Pivotal GemFire Native client 8.2.3](#).

In addition to bug fixes, **Pivotal GemFire Native Client 8.2.2** includes this new feature:

- Open SSL 1.0.1s is now supported.

In addition to bug fixes, **Pivotal GemFire Native Client 8.2.1** includes these new features and changes:

- Added support for Solaris 10 and 11 on 32-bit and 64-bit x86 architectures.
- **GEM-45**: The new update-locator-list-interval pool attribute controls the frequency of locator list updates.
- Open SSL 1.0.1s is now supported.

In addition to bug fixes, **Pivotal GemFire Native Client 8.2.0** includes these new features and changes:

- **Support for Java SE 8** (in addition to Java SE 7).
- **Production support for RHEL 7**. See [Pivotal GemFire Native Client Supported Configurations](#) for more information.
- **Deprecated Platforms**. In GemFire 8.2, support for the following platforms is deprecated and may be removed in a future release of the product:
  - All 32-bit operating systems.
  - RHEL 5
  - Mac OS X 10.8
  - Windows 2008 Server R2
  - Windows 7
  - Windows 8
  - Ubuntu 10
Installing Pivotal GemFire Native Client 8.2

You can download Pivotal GemFire Native Client 8.2.0 (and Pivotal GemFire 8.2.0) from the Pivotal GemFire product download page.

Pivotal GemFire Native Client 8.2 installers are available in ZIP (Linux, Solaris) and MSI (Windows) format.

For details on how to install Pivotal GemFire Native Client 8.2, see Installing the Native Client in the Pivotal GemFire Native Client User's Guide.
GemFire Native Client 8.2 Resolved Issues

This section lists issues resolved in GemFire Native Client 8.2, with the most recent release shown first. Ticket reference numbers come from the GemFire ticketing system.

Issues Resolved in Pivotal GemFire Native client 8.2.7

- GEMNC-414: Fixes internal timeout magnitude.

Issues Resolved in Pivotal GemFire Native client 8.2.6

- GEMNC-215: Time out in a deterministic fashion. Normalized the handling of time throughout the client.

Issues Resolved in Pivotal GemFire Native client 8.2.5

- GEMNC-391: Repaired a double free in `ThinClientRegion::executeFunctionSH` that could lead to a segmentation fault.

Issues Resolved in Pivotal GemFire Native client 8.2.4

- GEMNC-365: Eliminated throwing a spurious `gemfire::IllegalStateException: CacheableString::asChar` exception, which occurred even though the C++ application did not invoke `CacheableString::asChar()`.

Issues Resolved in Pivotal GemFire Native client 8.2.3

- GEMNC-355: Fixed an issue that caused orphaned worker threads.

Issues Resolved in Pivotal GemFire Native client 8.2.2

- GEMNC-30: Updates support rolling upgrades to Pivotal GemFire 8.2.2 server, without changing client-side code. Unmodified version 8.2.2 client-side code also works with Pivotal GemFire 9.0 servers.

Issues Resolved in Pivotal GemFire Native client 8.2.1

- GEM-191: Moved inline function definitions to later in the source for compilation correctness.
- GEM-336: Fixed an access violation in `HostStatHelperWin` when `ProcessObj` was NULL.
- GEM-145: Add support for two-phase commit transactions.
- GEM-533: Updated the SSL module to support OpenSSL 1.0.1s.
- GEM-898: The `removeAll` operation now correctly assigns sequence numbers based on the number of items removed. This fix is available in v8.2.1.1.
- GEM-766: Fixed a bug in the read-timeout handshake with a server. This fix is available in v8.2.1.1.
- GEMNC-145/GEM-898: The `removeAll` operation now correctly assigns sequence numbers based on the number of items removed.
- GEMNC-157/GEM-766: Corrects the read-timeout handshake with a server.
- GEMNC-108: Fixed a bug that could have resulted in a corrupted function's result collector. The bug was caused by error handling code operating outside of the guarded scope intended to modify the result collector.
- GEMNC-90: C++ signal handling is now done by the operating system, instead of going through a custom core dump handler. As a result, signal handling is more accurate.
• **GEMNC-117**: Memory usage is improved for `getASCIIString` when calculating the encoded length.
• **GEMNC-281**: Fixed buffer overflow in string serialization in the .NET client.

**Issues Resolved in Pivotal GemFire Native client 8.2.0**

• [#1026](#). utf char doesn't work inside cq/ query
• [#51902](#). Added support for CacheableLinkedList type.
Pivotal GemFire Native Client Supported Configurations

The Pivotal GemFire native client provides access for C++ and Microsoft® .NET™ clients to the GemFire distributed system. It operates on platforms running Microsoft Windows, Linux (Intel), and Sun Solaris.

Operating system requirements are listed in the chart below:

Note: GemFire 8.2 deprecates all 32-bit platforms and other platforms as indicated below and in the *Pivotal GemFire Native Client 8.2 Release Notes*. Support for deprecated platforms may be removed in a future release of the product.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Processor and Architecture</th>
<th>RAM</th>
<th>Swap Space</th>
<th>32-bit Disk Space Required</th>
<th>64-Bit Disk Space Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solaris 10</td>
<td>x86 (64-bit) (Deprecated in v8.2.1)</td>
<td>2GB</td>
<td>256MB</td>
<td></td>
<td>120MB</td>
</tr>
<tr>
<td>Solaris 11</td>
<td>x86 (64-bit)</td>
<td>2GB</td>
<td>256MB</td>
<td></td>
<td>120MB</td>
</tr>
<tr>
<td>RHEL 5 (Deprecated in v8.2)</td>
<td>x86 (64-bit)</td>
<td>2GB</td>
<td>256MB</td>
<td>105MB</td>
<td>120MB</td>
</tr>
<tr>
<td></td>
<td>x86 (32-bit) (Deprecated in v8.2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RHEL 6</td>
<td>x86 (64-bit)</td>
<td>2GB</td>
<td>256MB</td>
<td>105MB</td>
<td>120MB</td>
</tr>
<tr>
<td></td>
<td>x86 (32-bit) (Deprecated in v8.2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RHEL 7</td>
<td>x86 (64-bit)</td>
<td>2GB</td>
<td>256MB</td>
<td>105MB</td>
<td>120MB</td>
</tr>
<tr>
<td></td>
<td>x86 (32-bit) (Deprecated in v8.2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLES 10**</td>
<td>x86 (64-bit)</td>
<td>2GB</td>
<td>256MB</td>
<td>105MB</td>
<td>120MB</td>
</tr>
<tr>
<td></td>
<td>x86 (32-bit) (Deprecated in v8.2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Windows 2008</td>
<td>x86 (64-bit)</td>
<td>2GB</td>
<td>256MB</td>
<td>104MB</td>
<td>125MB</td>
</tr>
<tr>
<td>R2 (Deprecated in v8.2)</td>
<td>x86 (32-bit) (Deprecated in v8.2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Pivotal GemFire Native Client Supported Configurations**

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Processor and Architecture</th>
<th>RAM</th>
<th>Swap Space</th>
<th>32-bit Disk Space Required</th>
<th>64-Bit Disk Space Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 2012 Server**</td>
<td>x86 (64-bit)</td>
<td>2GB</td>
<td>256MB</td>
<td>104MB</td>
<td>125MB</td>
</tr>
<tr>
<td>(Deprecated in v8.2)</td>
<td>x86 (32-bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Deprecated in v8.2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Windows 2012 Server R2**</td>
<td>x86 (64-bit)</td>
<td>2GB</td>
<td>256MB</td>
<td>104MB</td>
<td>125MB</td>
</tr>
<tr>
<td></td>
<td>x86 (32-bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Deprecated in v8.2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Windows 8**</td>
<td>x86 (64-bit)</td>
<td>2GB</td>
<td>256MB</td>
<td>104MB</td>
<td>125MB</td>
</tr>
<tr>
<td>(Deprecated in v8.2)</td>
<td>x86 (32-bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Deprecated in v8.2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Windows 8.1**</td>
<td>x86 (64-bit)</td>
<td>2GB</td>
<td>256MB</td>
<td>104MB</td>
<td>125MB</td>
</tr>
<tr>
<td></td>
<td>x86 (32-bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Deprecated in v8.2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Indicates operating systems that have not been fully tested but are still supported.**

**Windows Support Details**

The Pivotal GemFire native client is built and tested on Windows 2008 R2.

The native client is not supported on Windows NT 4.0.

**Runtime Library Requirements**

The GemFire native client also requires the Microsoft Visual C++ 2010 Redistributable Package. This package is installed for you when you use the native client .msi installer. If you do not use the .msi installer, you will need to install this package for your platform architecture manually (x86 or x64) on all machines that will run the GemFire native client. This package contains runtime libraries needed by the native client.

**.NET Framework Version Support**

The Pivotal GemFire native client is supported with Microsoft .NET Framework 3.0, and 3.5 and 4.0.

Microsoft .NET Framework 2.0 is no longer supported.

A Microsoft .NET Framework must be installed to support the C++/CLI (Common Language Infrastructure) library for the native client.

**Note:** You can download the .NET Framework Redistributable Package (x86 for 32-bit or x64 for 64-bit) for the version you wish to use from [http://www.microsoft.com/downloads](http://www.microsoft.com/downloads). If it isn’t listed on the Download Center page, use the Search tool to search for ".NET Framework”.


**Linux**

The Pivotal GemFire native client is built on Red Hat Enterprise ES 5 kernel version 2.6.18-8.el5.
The native client is tested on the following Linux versions:

- Red Hat Enterprise 5 release 5 (Tikanga), kernel version 2.6.18-8.EL5
- Red Hat Enterprise 6, kernel version 2.6.32-220.el6.x86_64

If you are not sure of the kernel version on your system, use this command to list it:

```
prompt> uname -r
```

The following table lists the RPM package dependencies for several Linux distributions. The i386 or i686 after the package name indicates that you must install the package for that particular architecture regardless of the native operating system architecture. All of the packages listed are available with the default media for each distribution.

### Table 1: GemFire Dependencies on Linux RPM Packages

<table>
<thead>
<tr>
<th>Linux Version</th>
<th>glibc</th>
<th>libgcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red Hat Enterprise Linux Server release 5 (i686)</td>
<td>glibc</td>
<td>libgcc</td>
</tr>
<tr>
<td>Red Hat Enterprise Linux Server release 5 (x86_64)</td>
<td>glibc (i686)</td>
<td>libgcc (i386)</td>
</tr>
<tr>
<td>Red Hat Enterprise Linux Server release 6 (i686)</td>
<td>glibc</td>
<td>glibc</td>
</tr>
<tr>
<td>Red Hat Enterprise Linux Server release 6 (x86_64)</td>
<td>glibc (i686)</td>
<td>libgcc (i386)</td>
</tr>
</tbody>
</table>

For versions of Linux not listed in the table, you can verify that you meet the native client dependencies at the library level by using the `ldd` tool and entering this command:

```
prompt> ldd $GFCPP/lib/libgfcppcache.so
```

This step assumes you have already installed the native client and have set the GFCPP environment variable to `productDir`, where `productDir` represents the location of the `NativeClient_xxxx_bnnnnn` directory (`xxxx` is the four-digit product version and `bnnnnn` is the product build number).

The following libraries are external dependencies of the native library, `libgfcppcache.so`. Verify that the `ldd` tool output includes all of these:

- `libdl.so.2`
- `libm.so.6`
- `libpthread.so.0`
- `libc.so.6`
- `libz.so.1`

For details on the `ldd` tool, see its Linux online man page.

### Solaris

The Pivotal GemFire native client is supported on the following Solaris versions:

- Solaris 9 kernel update 118558-38
- Solaris 10 kernel update 118833-24
Running Pivotal GemFire Native Client on vSphere

Without tuning, the Pivotal GemFire native client can suffer a performance drop in virtual environments, including the VMware vSphere virtual platform. Without correct tuning, you can expect to see significant performance degradation when running the GemFire native client on vSphere versus running GemFire on dedicated hardware.

We recommend that you tune your GemFire native client on vSphere deployments using the same guidelines published for Pivotal GemFire.

Software Requirements for Using SSL

If you plan on using SSL in your GemFire native client and server deployment, you will need to download and install OpenSSL.

The GemFire native client requires OpenSSL 1.0.1s or later. For Windows platforms, you can use either the regular or OpenSSL "Light" version.

In addition, make sure that your system environment variables have been configured to include OpenSSL.
Part V

Getting Started with Pivotal GemFire Native Client

*Getting Started with Pivotal GemFire Native Client* gives you a conceptual overview of the Pivotal GemFire® native client. It shows you how to install the product, build native client applications on various platforms, and run the product examples.

The Pivotal GemFire native client provides access for C++ and Microsoft® .NET™ clients to the Pivotal GemFire distributed system.

About the Native Client

The native client delivers the full set of capabilities supplied by Java clients communicating with the GemFire cache server.

The native client is written entirely in C++, so its initialization process does not involve the creation of a Java virtual machine. The .NET native client provides native operations for the .NET Framework application developer who writes in .NET languages and needs to access the GemFire cache server.

Pivotal GemFire native clients in C++, Java, and .NET languages communicate only with the cache server and do not communicate with each other. The native clients interface with the server at the sockets level and implement the same wire protocol to the server. These capabilities produce extremely high performance and system scalability.

C++ and .NET native clients provide access to the full region API, including support for application plug-ins, managed connectivity, highly available data, and reliable failover to a specified server list. All of this is transparent to the end user.

You can configure GemFire native clients to cache data locally, or they can act in a cacheless mode where they retrieve data from a cache server and directly pass it to other system members without incurring the caching overhead. They can be configured as read only caches, or be configured to receive notifications from the server whenever a key of interest to the client changes on the server.

This is a conceptual overview of how .NET and C++ applications access the cache server.
Installing the Native Client

Procedures for installing the native client vary according to your operating system.
Installing on Linux and Solaris

You install the native client on Linux and Solaris by extracting the contents of a ZIP file. After installation, you add environment settings.

Installation Prerequisites

Before you install the Pivotal GemFire native client, you must complete the following prerequisites:

- Confirm that your system meets the hardware and software requirements described in Pivotal GemFire Native Client Supported Configurations.
- From the Pivotal GemFire download page, select Download.
- Under File Groups, select and download the Pivotal GemFire native client .zip file appropriate for your operating system and hardware architecture.
  - For Linux platforms, download Pivotal_GemFire_NativeClient_Linux_architecture_xxxx_bNNNN.zip where architecture equals either 32bit or 64bit, xxxx represents the four-digit version of the product and bNNNN corresponds to the build number for the release.
  - For Solaris platforms, download Pivotal_GemFire_NativeClient_Solaris_architecture_xxxx_bNNNN.zip where architecture equals either 32bit or 64bit, xxxx represents the four-digit version of the product and bNNNN corresponds to the build number for the release.

Installation Path for Linux

The default installation path is NativeClient_Linux_architecture_xxxx_bNNNN and reflects the architecture, product version and build number identified in the filename of the installer .zip file.

Installation Path for Solaris

The default installation path is NativeClient_Solaris_architecture_xxxx_bNNNN and reflects the architecture, product version and build number identified in the filename of the installer .zip file.

Environment Settings

When the installation is complete, you add these environment settings:

- Set the GFCPP environment variable to productDir, where productDir represents the NativeClient__platform_architecture_xxxx directory (where platform is the OS you are installing upon, architecture is 32bit or 64bit, and xxxx is the four-digit product version)
  - Add $GFCPP/bin to the PATH
  - Add $GFCPP/lib to the LD_LIBRARY_PATH.
Installing on Windows

Install the native client by using the Windows installer GUI interface or the command-line.

This installation topic contains the following sections:

- Prerequisites
- Install with the Windows Installer
- Install from the MSI Command-Line
- Repair a Windows Installation

Prerequisites

Before you install the Pivotal GemFire native client, you must complete the following prerequisites:

- Confirm that your system meets the hardware and software requirements described in Windows Support Details.
- From the Pivotal GemFire download page, select Download.
- On the Pivotal GemFire product download page, select and download the Pivotal GemFire native client .msi file appropriate for your hardware architecture.

Note: If you do not install the native client using the .msi installer or the msiexec command-line, then you will need to manually perform the configuration steps (such as configure your GFCPP and PATH environment variables) and install the Microsoft Visual C++ 2010 Redistributable Package described in the next section.

Install with the Windows Installer

The native client can be installed on Windows by using the Pivotal_GemFire_NativeClient_architecture_xxxx_bNNNN.msi Windows installer, where architecture is 32bit or 64bit, xxxx represents the four-digit version of the product and bNNNN corresponds to the build number for the release. The installer requires msiexec version 3.0 or higher.

Note: You must be logged in with administrative privileges or the MSI installation will fail.

Double-click the MSI file to start the installation. You are prompted to enter the installation location of the native client.

To complete the installation, the MSI installer automatically configures these native client system environment settings:

- Sets the GFCPP environment variable to productDir, where productDir represents the location where you installed the native client.
- Adds %GFCPP%\bin to the Windows PATH.
- Installs Microsoft Visual C++ 2010 Redistributable Package for your platform architecture (x86 or x64). This package contains runtime libraries required by the native client.

Install from the MSI Command-Line

In addition to the standard Windows installer interface, the Pivotal_GemFire_NativeClient_architecture_xxxx_bNNNN.msi file provides a set of command-line installer options. This is an example of the installer command-line syntax:

```bash
msiexec /q /i Pivotal_GemFire_NativeClient_architecture_xxxx_bNNNN.msi DEFAULT_INSTALLDIR=<path>
```
The following table lists common `msiexec` command-line options for use with `NativeClient_architecture_xxxx.msi`, along with an explanation of their usage. For a list of all command-line options, enter `msiexec/?`.

Table 2: MSI Command-Line Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>/q</td>
<td>Creates a quiet installation with no interface or prompts.</td>
</tr>
<tr>
<td>/i</td>
<td>Indicates that the product is to be installed or configured.</td>
</tr>
<tr>
<td>DEFAULT_INSTALLDIR=&lt;path&gt;</td>
<td>Specifies the destination directory, if different from the default.</td>
</tr>
<tr>
<td>/x</td>
<td>Indicates a product uninstall procedure.</td>
</tr>
</tbody>
</table>

### Repair a Windows Installation

If problems occur with your Windows native client installation you can initiate a repair operation to restore any missing elements or registry settings.

1. Double-click the MSI file, then click Next in the Setup Wizard screen.
2. In the following screen, click the Repair button.
3. In the next screen, click the Repair button.
Uninstalling the Native Client

How you uninstall the native client depends on your operating system.

Uninstall on Linux and Solaris
On Linux and Solaris, delete the `NativeClient_Linux_architecture_2023_bNNNN` product directory or the `NativeClient_Solaris_architecture_2023_bNNNN` product directory and all of its subdirectories.

Uninstall on Windows
Use the MSI installer graphical interface, its command-line interface, or the Windows Control Panel.

Uninstall from MSI Installer Graphical Interface
1. Double-click the MSI file, then click Next in the Setup Wizard screen.
2. In the following screen, click the Remove button.
3. In the next screen, click the Remove button and then click Finish.

Uninstall from MSI Installer Command-Line
This is the MSI command-line syntax for a quiet uninstall:

```
msiexec /q /x Pivotal_GemFire_NativeClient_architecture_2023_bNNNN.msi
DEFAULT_INSTALLDIR=<path>
```

Table 2: MSI Command-Line Options lists common `msiexec` command-line options for use with Pivotal GemFire NativeClient `architecture_2023_bNNNN.msi`, along with an explanation of their usage. For a list of all command-line options, enter `msiexec/?`.

The `/x` option uninstalls the product.

Uninstall from the Control Panel
1. From the Control Panel, open Add or Remove Programs.
2. Select **Pivotal GemFire Native Client [32bit or 64bit] [x.x.x.x]** (with the values appropriate for your installation.)
3. Click the Remove button.

Running Native Client Applications

Set up the environment for the native client on multiple platforms. Compile and run client programs.
Developing C++ Programs on Linux

This section describes how to build and run a native client application on Linux.

**Note:** When compiling external projects or applications that are used or referenced by the GemFire native client, make sure that you compile them for the same target architecture as your native client installation. For example, if you installed the 32-bit (x86) version of the native client, compile your external projects for 32-bit (x86) architecture.

**Step 1: Set Environment Variables**

Set the native client environment variables on each Linux host. For each case, `<productDir>` is the path to the native client product directory.

For Bourne and Korn shells (sh, ksh, bash)

```bash
GFCPP=<productDir>; export GFCPP
PATH=$GFCPP/bin:$PATH; export PATH
LD_LIBRARY_PATH=$GFCPP/lib:$LD_LIBRARY_PATH; export LD_LIBRARY_PATH
```

**Step 2: Compile C++ Clients and Dynamically Link Them to the GemFire Library**

On Linux, the `g++` compiler is supported. To build and link a C++ client to GemFire on Linux, the compilation command line must include the arguments listed in the following table.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-D_REENTRANT</code></td>
<td>Required to compile Linux programs in a thread-safe way.</td>
</tr>
<tr>
<td><code>-m32</code></td>
<td>Enables 32-bit compilation.</td>
</tr>
<tr>
<td><code>-m64</code></td>
<td>Enables 64-bit compilation.</td>
</tr>
<tr>
<td><code>-I$GFCPP/include</code></td>
<td>Specifies the native client include directory.</td>
</tr>
</tbody>
</table>

The following table lists the linker switches that must be present on the command line when dynamically linking to the GemFire library.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-rpath $GFCPP/lib</code></td>
<td>Tells the linker to look in <code>$GFCPP/lib</code> for libraries on which the GemFire library depends.</td>
</tr>
<tr>
<td><code>-L$GFCPP/lib</code></td>
<td>Tells the linker where to find the named libraries.</td>
</tr>
<tr>
<td><code>-o durableclient</code></td>
<td>Tells the linker to output an object file named 'durableclient'.</td>
</tr>
<tr>
<td><code>-lgfcppcache</code></td>
<td>Links the GemFire C++ cache library to the compiled executable.</td>
</tr>
</tbody>
</table>

The following examples compile and link the `$GFCPP/SampleCode/quickstart/cpp/DurableClient.cpp` client to the `durableclient` output file.
Compiling and Dynamically Linking on Linux for 32-bit

```bash
g++ \
-D_REENTRANT \n-03 \n-Wall \n-m32 \n-Is$GFCPP/include \n cpp/DurableClient.cpp \n cpp/plugins/DurableCacheListener.cpp \n -o cpp/DurableClient \n -L$GFCPP/lib \n -Wl,-rpath,$GFCPP/lib \n -lgfccppcache
```

Compiling and Dynamically Linking on Linux for 64-bit

```bash
g++ \
-D_REENTRANT \n-03 \n-Wall \n-m64 \n-Is$GFCPP/include \n cpp/DurableClient.cpp \n cpp/plugins/DurableCacheListener.cpp \n -o cpp/DurableClient \n -L$GFCPP/lib \n -Wl,-rpath,$GFCPP/lib \n -lgfccppcache
```

Step 3: Make Sure the GemFire Library Can Be Loaded

When the C++ application is dynamically linked to the GemFire library, the library must be dynamically loadable.

To ensure that the GemFire library is available for loading, make sure you have added the path $GemFireDir/lib to the `LD_LIBRARY_PATH` environment variable, where $GemFireDir is the path to the GemFire product directory.
Developing C++ Programs on Solaris

This section describes how to build and run a native client application on Solaris.

**Note:** When compiling external projects or applications that are used or referenced by the GemFire native client, make sure that you compile them for the same target architecture as your native client installation. For example, if you installed the 32-bit (x86) version of the native client, compile your external projects for 32-bit (x86) architecture.

**Step 1. Set Environment Variables**

Set the native client environment variables on each Solaris host. For each case, `<productDir>` is the path to the native client product directory.

For Bourne and Korn shells (sh, ksh, bash)

```bash
GFCPP=<productDir>; export GFCPP
PATH=$GFCPP/bin:$PATH; export PATH
LD_LIBRARY_PATH=$GFCPP/lib:$LD_LIBRARY_PATH; export LD_LIBRARY_PATH
```

**Step 2. Compile C++ Clients and Dynamically Link to Them to GemFire Library**

Version 5.9 of the SUNpro compiler is supported on Solaris. The linker switches vary according to whether you are statically linking to the GemFire library.

To build and link a C++ client to GemFire on Solaris, the compilation command line must include the appropriate arguments from this table.

**Table 5: Compiler Arguments on Solaris**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-D_REENTRANT</code></td>
<td>Required to compile Solaris programs in a thread-safe way.</td>
</tr>
<tr>
<td><code>-xarch=v8plus</code></td>
<td>Enables 32-bit compilation.</td>
</tr>
<tr>
<td><code>-xarch=v9</code></td>
<td>Enables 64-bit compilation.</td>
</tr>
<tr>
<td><code>-ldl;-lpthread;-lc;-lm;-lsocket;-lrt;-lnsl;-ldemangle;-lkstat;-lz</code></td>
<td>Additional libraries.</td>
</tr>
<tr>
<td><code>-library=stlport4</code></td>
<td>Solaris library compilation.</td>
</tr>
<tr>
<td><code>-I$ GFCPP /include</code></td>
<td>Specifies the GemFire include directory.</td>
</tr>
</tbody>
</table>

**Step 3. Make Sure the GemFire Library Can Be Loaded**

When a C++ application is not statically linked to the GemFire library, the library must be dynamically loadable.

To verify that the GemFire library is available for loading, make sure you have added the path `GemFireDir/lib` to the `LD_LIBRARY_PATH` environment variable, where `GemFireDir` is the path to the GemFire product directory.
Developing C++ Programs on Windows

GemFire uses the Visual Studio 2010 Service Pack 1 compiler for C++ programs on Windows, which invokes Microsoft® cl.exe from the command line at compile time.


Visual Studio 2010 SP1 is the recommended compiler. If you are using any other compiler, contact technical support for assistance.

Note: When compiling external projects or applications that are used or referenced by the GemFire native client, make sure that you compile them for the same target architecture as your native client installation. For example, if you installed the 32-bit (x86) version of the native client, compile your external projects for 32-bit (x86) architecture.

Step 1. Installer Sets Up Environment Variables

When you install on Windows, the installer performs these tasks:

• Sets the GFCPP environment variable to <productDir>, where <productDir> is the path to the native client product directory.
• Adds the %GFCPP%\bin executable directory to the Windows PATH.

Step 2. Choose 32-bit or 64-bit Command-line Prompt

For 32-bit:
Start > Programs > Microsoft Visual Studio 2010 > Visual Studio Tools > Visual Studio 2010 Command Prompt

For 64-bit:
Start > Programs > Microsoft Visual Studio 2010 > Visual Studio Tools > Visual Studio 2010 x64 Win64 Command Prompt

To build using the Microsoft Visual Studio Interface, from the Solutions Platform, choose Win32 or x86 from the Build menu for 32-bit builds or x64 for a 64-bit build.

Step 3. Compile C++ Clients and Dynamically Link Them to GemFire Library

The following table lists the compiler and linker switches that must be present on the cl.exe command line.

Note: If you want to use the Visual Studio user interface instead of invoking cl.exe from the command line, be sure to supply these parameters.

Table 6: Compiler and Linker Switches for Windows

<table>
<thead>
<tr>
<th>Argument</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>/MD</td>
<td>Memory model.</td>
</tr>
<tr>
<td>/EHsc</td>
<td>Catches C++ exceptions only and tells the compiler to assume that <em>extern</em> C functions never throw a C++ exception.</td>
</tr>
</tbody>
</table>
### Argument | Explanation
---|---
/GR | Runtime type information.
-I%GFCPP%\include | Specifies the GemFire include directory.
%GFCPP%\lib\gfccache.lib | Specifies the library file for the shared library.

### Step 4. Verify that You Can Load the GemFire Library
Because GemFire does not provide a library that can be linked statically into an application on Windows, you must dynamically link to the GemFire library.

To make the GemFire library available for loading, verify that the directory `GemFireDir/bin` is included in the `PATH` environment variable, where `GemFireDir` is the path to the GemFire product directory.

### QuickStart Examples and Guide
Run the GemFire native client QuickStart examples and read the QuickStart Guide to understand native client functionality.
Running the QuickStart Examples

The QuickStart examples demonstrate the capabilities of the GemFire native client, and provide source code so you can examine how each example is designed. C++ and C# examples demonstrate how the native client performs as a C++ or C# client.

Read the QuickStart Guide first (the README.HTML file included with the QuickStart examples) to learn more about running the QuickStart examples.

QuickStart examples are packaged as part of the native client product download listed on the Pivotal GemFire download page. Product examples are installed into the following directory when you install the Pivotal GemFire native client:

```
<native_client_installdir>/SampleCode/quickstart
```

where `native_client_installdir` is the location where you installed the Pivotal GemFire native client.

**Note:** The C# examples have a .config file that should not be modified or deleted. The file provides references to support files for running the example.
Accessing the QuickStart Guide

The QuickStart Guide for the native client consists of a set of compact programming samples that demonstrate both C++ and C# client operations.

Open the QuickStart Guide to rapidly become familiar with native client functionality. To view the QuickStart Guide, navigate to the SampleCode/quickstart directory of your Pivotal GemFire native client installation, and open the README.html file.

The README.html file describes each programming sample, explains initial environment setup, and provides instructions for running the QuickStart examples.
Part VI

Setting System Properties

Setting System Properties describes how to configure Pivotal GemFire native clients and cache servers to participate in a distributed system.

Configuring the Native Client and Cache Server

You can configure clients through files and API calls. The servers are configured through command-line input and configuration files.
## Native Client Configuration

You configure the native client in two files: `gfcpp.properties` for native client system-level configuration and `cache.xml` for cache-level configuration.

The configuration of the caches is part of the application development process. See *Cache Initialization File*. (The cache-level configuration file is generally referred to as `cache.xml`, but you can use any name.)

- About `gfcpp.properties` Configuration File
- Configuration File Locations
- Configuring System Properties for the Native Client
- Running a Native Client Out of the Box

### About `gfcpp.properties` Configuration File

The `gfcpp.properties` file provides local settings required to connect a client to a distributed system, along with settings for licensing, logging, and statistics. See *Attributes in gfcpp.properties*.

The application software may include a set of `gfcpp.properties` files. You set any attributes needed for the application design in these files, then you can add any attributes needed for the local site.

If you do not have `gfcpp.properties` files, use any text editor to create them. See `gfcpp.properties Example File` for a sample of the file format and contents.

### Configuration File Locations

A native client looks for `gfcpp.properties` first in the working directory where the process runs, then in `productDir/defaultSystem`. Use the `defaultSystem` directory to group configuration files or to share them among processes for more convenient administration. If `gfcpp.properties` is not found, the process starts up with the default settings.

For the `cache.xml` cache configuration file, a native client looks for the path specified by the `cache-xml-file` attribute in `gfcpp.properties` (see *Attributes in gfcpp.properties*). If the `cache.xml` is not found, the process starts with an unconfigured cache.

### Configuring System Properties for the Native Client

The typical configuration procedure for a native client includes the high-level steps listed below. The rest of this chapter provides the details.

1. Place the `gfcpp.properties` file for the application in the working directory or in `productDir/defaultSystem`. Use the configuration file that came with the application software if there is one, or create your own. See `gfcpp.properties Example File` for a sample of the file format and contents.
2. Place the `cache.xml` file for the application in the desired location and specify its path in the `gfcpp.properties` file.
3. Add other attributes to the `gfcpp.properties` file as needed for the local system architecture. See *Attributes in gfcpp.properties* for the configurable attributes, and `gfcpp.properties Example File` for a sample of the file format.

### Running a Native Client Out of the Box

If you start a native client without any configurations, it uses any attributes set programmatically plus any hard-coded defaults (listed in *Attributes in gfcpp.properties*). Running with the defaults is a convenient way to learn the operation of the distributed system and to test which attributes need to be reconfigured for your environment.
Cache Server Configuration

You configure the cache server in two files: `gemfire.properties` for server system-level configuration and `cache.xml` for cache-level configuration.

The configuration of the caches is part of the application development process. See Cache Initialization File. (The cache-level configuration file is generally referred to as `cache.xml`, but you can use any name.)

Configuration File Locations

For the GemFire cache server, the `gemfire.properties` file is usually stored in the current working directory. For more information, see the GemFire User’s Guide.

For the `cache.xml` cache configuration file, a native client looks for the path specified by the `cache-xml-file` attribute in `gfcpp.properties` (see Attributes in `gfcpp.properties`). If the `cache.xml` is not found, the process starts with an unconfigured cache.

Modifying Attributes Outside the `gemfire.properties` File

In addition to the `gemfire.properties` file, you can pass attributes to the cache server on the `gfsh` command line. These override any settings found in the `gemfire.properties` file when starting the cache server.

For more information, see Configuring a Pivotal GemFire Cluster in the GemFire User's Guide.
Attribute Definition Priority

You can specify attributes in different ways, which can cause conflicting definitions. Applications can be configured programmatically, and that has priority over other settings.

Check your application documentation to see whether this applies in your case.

- Programmatic configuration
- Properties set at the command line
  - `currentWorkingDirectory/gfccpp.properties` file
  - `productDir/defaultSystem/gfccpp.properties` file
- GemFire defaults

In case an attribute is defined in more than one place, the first source in this list is used:

The `gfccpp.properties` files and programmatic configuration are optional. If they are not present, no warnings or errors occur. For details on programmatic configuration through the `Properties` object, see "Defining Properties Programmatically."

For information on the GemFire cache server configuration, see the *GemFire User's Guide*. 
Search Path for Multiple gfcpp.properties Files

The native client and cache server processes first look for their properties file in the `productDir/defaultSystem` directory, then in the working directory.

Any properties set in the working directory override settings in the `defaultSystem/gfcpp.properties` file.

If you are running multiple processes on one machine, you can configure the `gfcpp.properties` file in the `defaultSystem` directory as a shared file that all processes can find. If a few processes need a slightly different configuration, you can put individual `gfcpp.properties` files in their home directories to override specific properties.
Overriding gfcpp.properties Settings

Application developers have the option of configuring system attributes programmatically, rather than using the gfcpp.properties file.

Attributes set programmatically override any matching attribute settings in the gfcpp.properties file, but additional attributes not set programmatically will be configured using the settings in gfcpp.properties.
Defining Properties Programmatically

You can pass in specific gfcpp properties programmatically by using a `Properties` object to define the non-default properties.

Example:

```cpp
PropertiesPtr systemProps = Properties::create();
systemProps->insert( "statistic-archive-file", "stats.gfs" );
systemProps->insert( "cache-xml-file", ".\myapp-cache.xml" );
systemProps->insert( "stacktrace-enabled", "true" );
CacheFactoryPtr systemPtr = CacheFactory::createCacheFactory(systemProps);
```

Attributes in gfcpp.properties

A variety of `gfcpp.properties` settings can be used when a native client connects to a distributed system. The following settings can be configured:

- **General Properties** Basic information for the process, such as cache creation parameters.
- **Logging Properties** How and where to log system messages.
- **Statistics Archiving Properties** How to collect and archive statistics information.
- **Durable Client Properties** Information about the durable clients connected to the system.
- **Security Properties** Information about various security parameters.

Attribute Definitions

The following tables list GemFire configuration attributes that can be stored in the `gfcpp.properties` file to be read by a native client.

For the system properties that relate to high availability, see **Sending Periodic Acknowledgement**. For a list of security-related system properties and their descriptions, see the table **System Properties for Client Authentication and Authorization**.

Table 7: Attributes in gfcpp.properties: General Properties

<table>
<thead>
<tr>
<th><code>gfcpp.properties</code> Attribute</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>appdomain-enabled</td>
<td>If <code>true</code>, allows native client to work when multiple .NET appdomains are in use.</td>
<td>false</td>
</tr>
<tr>
<td>cache-xml-file</td>
<td>Name and path of the file whose contents are used by default to initialize a cache if one is created. If not specified, the native client starts with an empty cache, which is populated at runtime. See <a href="#">Cache Initialization File</a> for more information on the cache initialization file.</td>
<td>no default</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>heap-lru-delta</td>
<td>When heap LRU is triggered, this is the amount that gets added to the percentage that is above the heap-lru-limit amount. LRU continues until the memory usage is below heap-lru-limit minus this percentage. This property is only used if heap-lru-limit is greater than 0.</td>
<td>10</td>
</tr>
<tr>
<td>heap-lru-limit</td>
<td>Maximum amount of memory, in megabytes, used by the cache for all regions. If this limit is exceeded by heap-lru-delta percent, LRU reduces the memory footprint as necessary. If not specified, or set to 0, memory usage is governed by each region's LRU entries limit, if any.</td>
<td>0</td>
</tr>
<tr>
<td>conflate-events</td>
<td>Client side conflation setting, which is sent to the server.</td>
<td>server</td>
</tr>
<tr>
<td>connect-timeout</td>
<td>Amount of time (in seconds) to wait for a response after a socket connection attempt.</td>
<td>59</td>
</tr>
<tr>
<td>connection-pool-size</td>
<td>Number of connections per endpoint</td>
<td>5</td>
</tr>
<tr>
<td>crash-dump-enabled</td>
<td>Whether crash dump generation for unhandled fatal errors is enabled. True is enabled, false otherwise.</td>
<td>true</td>
</tr>
<tr>
<td>disable-chunk-handler-thread</td>
<td>When set to false, each application thread processes its own response. If set to true, the chunk-handler-thread processes the response for each application thread.</td>
<td>false</td>
</tr>
<tr>
<td>disable-shuffling-of-endpoints</td>
<td>If true, prevents server endpoints that are configured in pools from being shuffled before use.</td>
<td>false</td>
</tr>
<tr>
<td>grid-client</td>
<td>If true, the client does not start various internal threads, so that startup and shutdown time is reduced.</td>
<td>false</td>
</tr>
<tr>
<td>max-fe-threads</td>
<td>Thread pool size for parallel function execution. An example of this is the GetAll operations.</td>
<td>2 * number of CPU cores</td>
</tr>
</tbody>
</table>
### gfcpp.properties Attribute Description Default

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>max-socket-buffer-size</td>
<td>Maximum size of the socket buffers, in bytes, that the native client will try to set for client-server connections.</td>
<td>65 * 1024</td>
</tr>
<tr>
<td>notify-ack-interval</td>
<td>Interval, in seconds, in which client sends acknowledgments for subscription notifications.</td>
<td>1</td>
</tr>
<tr>
<td>notify-dupcheck-life</td>
<td>Amount of time, in seconds, the client tracks subscription notifications before dropping the duplicates.</td>
<td>300</td>
</tr>
<tr>
<td>ping-interval</td>
<td>Interval, in seconds, between communication attempts with the server to show the client is alive. Pings are only sent when the ping-interval elapses between normal client messages. This must be set lower than the server’s maximum-time-between-pings.</td>
<td>10</td>
</tr>
<tr>
<td>redundancy-monitor-interval</td>
<td>Interval, in seconds, at which the subscription HA maintenance thread checks for the configured redundancy of subscription servers.</td>
<td>10</td>
</tr>
<tr>
<td>stacktrace-enabled</td>
<td>If true, the exception classes capture a stack trace that can be printed with their printStackTrace function. If false, the function prints a message that the trace is unavailable.</td>
<td>false</td>
</tr>
<tr>
<td>tombstone-timeout</td>
<td>Time in milliseconds used to timeout tombstone entries when region consistency checking is enabled. Note: Do not modify this property unless instructed by technical support.</td>
<td>480000</td>
</tr>
</tbody>
</table>

Table 8: Attributes in gfcpp.properties: Logging Properties

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>log-disk-space-limit</td>
<td>Maximum amount of disk space, in megabytes, allowed for all log files, current, and rolled. If set to 0, the space is unlimited.</td>
<td>0</td>
</tr>
</tbody>
</table>
### gfcpp.properties Attribute

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>log-file</td>
<td>Name and full path of the file where a running client writes log messages. If not specified, logging goes to <code>stdout</code>.</td>
<td>no default file</td>
</tr>
<tr>
<td>log-file-size-limit</td>
<td>Maximum size, in megabytes, of a single log file. Once this limit is exceeded, a new log file is created and the current log file becomes inactive. If set to 0 , the file size is unlimited.</td>
<td>0</td>
</tr>
<tr>
<td>log-level</td>
<td>Controls the types of messages that are written to the application's log. These are the levels, in descending order of severity and the types of message they provide:</td>
<td>config</td>
</tr>
<tr>
<td></td>
<td>• <strong>Error</strong> (highest severity) is a serious failure that will probably prevent program execution.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <strong>Warning</strong> is a potential problem in the system.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <strong>Info</strong> is an informational message of interest to the end user and system administrator.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <strong>Config</strong> is a static configuration message, often used to debug problems with particular configurations.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <strong>Fine, Finer, Finest, and Debug</strong> provide tracing information. Only use these with guidance from technical support.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Enabling logging at any level enables logging for all higher levels.</td>
<td></td>
</tr>
</tbody>
</table>

Table 9: Attributes in gfcpp.properties: Statistics Archiving Properties

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>statistic-sampling-enabled</td>
<td>Controls whether the process creates a statistic archive file.</td>
<td>true</td>
</tr>
<tr>
<td><code>gfcpp.properties</code> Attribute</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>statistic-archive-file</td>
<td>Name and full path of the file where a running system member writes archives statistics. If <code>archive-disk-space-limit</code> is not set, the native client appends the process ID to the configured file name, like <code>statArchive-PID.gfs</code>. If the space limit is set, the process ID is not appended but each rolled file name is renamed to <code>statArchive-ID.gfs</code>, where ID is the rolled number of the file.</td>
<td>./statArchive.gfs</td>
</tr>
<tr>
<td>archive-disk-space-limit</td>
<td>Maximum amount of disk space, in megabytes, allowed for all archive files, current, and rolled. If set to 0, the space is unlimited.</td>
<td>0</td>
</tr>
<tr>
<td>archive-file-size-limit</td>
<td>Maximum size, in bytes, of a single statistic archive file. Once this limit is exceeded, a new statistic archive file is created and the current archive file becomes inactive. If set to 0, the file size is unlimited.</td>
<td>0</td>
</tr>
<tr>
<td>statistic-sample-rate</td>
<td>Rate, in seconds, that statistics are sampled. Operating system statistics are updated only when a sample is taken. If statistic archival is enabled, then these samples are written to the archive. Lowering the sample rate for statistics reduces system resource use while still providing some statistics for system tuning and failure analysis. You can view archived statistics with the optional VSD utility.</td>
<td>1</td>
</tr>
<tr>
<td>enable-time-statistics</td>
<td>Enables time-based statistics for the distributed system and caching. For performance reasons, time-based statistics are disabled by default. See <code>System Statistics</code>.</td>
<td>false</td>
</tr>
</tbody>
</table>
Use the gfcpp.properties file to configure distributed system connections for the Pivotal GemFire native client.

The following example shows the format of a gfcpp.properties file. The first two attributes in this example should be set by programmers during application development, while other attributes are set on-site during system integration. The properties and their default settings that can be set in this file are described in detail in Attributes in gfcpp.properties.
Using the Default Sample File

A sample gfcpp.properties file is included with the Pivotal GemFire native client installation in the productDir/defaultSystem directory.

To use this file:

1. Copy the file to the directory where you start the application.
2. Uncomment the lines you need and edit the settings as shown in this example:

   ```
   cache-xml-file=test.xml
   ```

3. Start the application.

Default gfcpp.properties File

```
# Default Gemfire C++ distributed system properties
# Copy to current directory and uncomment to override defaults.
#
# Debugging support, enables stacktraces in gemfire::Exception.
# The default is false, uncomment to enable stacktraces in exceptions.
# stacktrace-enabled=true
# crash-dump-enabled=true
#
# Cache region configuration
# cache-xml-file=cache.xml
#
# Log file config
# log-file=gemfire_cpp.log
# log-level=config
# zero indicates use no limit.
# log-file-size-limit=0
# zero indicates use no limit.
# log-disk-space-limit=0
#
# Statistics values
# the rate is in seconds.
# statistic-sample-rate=1
# statistic-sampling-enabled=true
# statistic-archive-file=statArchive.gfs
# zero indicates use no limit.
# archive-file-size-limit=0
# zero indicates use no limit.
# archive-disk-space-limit=0
# enable-time-statistics=false
#
# Heap based eviction configuration
# maximum amount of memory used by the cache for all regions, 0 disables this feature
# heap-lru-limit=0
# percentage over heap-lru-limit when LRU will be called.
# heap-lru-delta=10
#
# Durable client support
# durable-client-id=
# durable-timeout=300
#
# SSL socket support
#
```
#ssl-enabled=false
#ssl-keystore=
#ssl-truststore=

## .NET AppDomain support
#appdomain-enabled=false

## Misc
#conflate-events=server
#disable-shuffling-of-endpoints=false
#grid-client=false
#max-fe-threads=
#max-socket-buffer-size=66560
# the units are in seconds.
#connect-timeout=59
#notify-ack-interval=10
#notify-dupcheck-life=300
#ping-interval=10
#redundancy-monitor-interval=10
#auto-ready-for-events=true

## module name of the initializer pointing to sample implementation from templates/security
#security-client-auth-library=securityImpl

## static method name of the library mentioned above
#security-client-auth-factory=createUserPasswordAuthInitInstance

## credential for Dummy Authenticator configured in server. note: security-password property will be inserted by the initializer mentioned in the above property.
#security-username=root
Configuring the Native Client Cache

Configuring the Native Client Cache describes the Pivotal GemFire native client cache functionality, initialization file, and APIs. It provides information about creating and working with caches, cache regions, and region entries. The Pivotal GemFire native client cache provides a framework for native clients to store, manage, and distribute application data.

Caches

The cache is the entry point to native client data caching in GemFire. Through the cache, native clients gain access to the GemFire caching framework for data loading, distribution, and maintenance.
About the Native Client Cache

The cache consists of data regions, each of which can contain any number of entries. Region entries hold the cached data. Every entry has a key that uniquely identifies it within the region and a value where the data object is stored.

The Cache instance allows your process to set general parameters for communication between a cache and other caches in the distributed system, and to create and access any region in the cache.

Regions are created from the Cache instance. Regions provide the entry point to the interfaces for instances of Region and RegionEntry.

Main Features and Functionality

The native client cache provides the following features and functionality.

- Local and distributed data caching for fast access.
- Data distribution between applications on the same or different platforms.
- Local and remote data loading through application plug-ins.
- Application plug-ins for synchronous and asynchronous handling of data events.
- Automated and application-specific data eviction for freeing up space in the cache, including optional overflow to disk.
- System message logging, and statistics gathering and archiving.

For more information specific to your client programming language, see the online API documentation and Pivotal GemFire Native Client C++ API and Pivotal GemFire Native Client .NET Caching API.
Cache APIs

The native client has two cache APIs, RegionService and Cache.

RegionService API

RegionService provides:

- Access to existing cache regions.
- Access to the standard query service for the cache, which sends queries to the servers. See Remote Querying and Continuous Querying.

RegionService is inherited by Cache.

You do not use instances of RegionService except for secure client applications with many users. See Creating Multiple Secure User Connections with RegionService.

Cache API

Use the Cache to manage your client caches. You have one Cache per client.

The Cache inherits RegionService and adds management of these client caching features:

- Region creation.
- Subscription keepalive management for durable clients.
- Access to the underlying distributed system.
- RegionService creation for secure access by multiple users.
Local, Remote, and Distributed Caches

The distributed system defines how native client and cache server processes find each other.

The distributed system keeps track of its membership list and makes its members aware of the identities of the other members in the distributed system.

A cache for a native client is referred to as its local cache. All other caches in the distributed system are considered remote caches to the application. Every cache server and application process has its own cache. The term distributed cache is used to describe the union of all caches in a GemFire distributed system.
Creating and Accessing a Cache

When you create a native client cache, you are creating a native client cache instance. You must provide some basic configuration information such as a connection name and cache initialization parameters for the native client's cache instance.

When you create a cache, you provide the following input:

- **Connection name.** Used in logging to identify both the distributed system connection and the cache instance. If you do not specify a connection name, a unique (but non-descriptive) default name is assigned.

- **cache.xml** to initialize the cache (if the initialization is not done programmatically). To modify the cache structure, you just edit cache.xml in your preferred text editor. No changes to the application code are required. If you do not specify a cache initialization file, you need to initialize the cache programmatically.

The cache.xml file contains XML declarations for cache, region, and region entry configuration.

This XML declares server connection pools and regions:

```xml
<cache>
  <region name="clientRegion1" refid="PROXY">
    <region-attributes pool-name="serverPool1"/>
  </region>
  <region name="clientRegion2" refid="PROXY">
    <region-attributes pool-name="serverPool2"/>
  </region>
  <region name="localRegion3" refid="LOCAL"/>
  <pool name="serverPool1">
    <locator host="host1" port="40404"/>
  </pool>
  <pool name="serverPool2">
    <locator host="host2" port="40404"/>
  </pool>
</cache>
```

When you use the regions, the client regions connect to the servers through the pools named in their configurations.

This file can have any name, but is generally referred to as cache.xml.

For a list of the parameters in the cache.xml file, including the DTD, see Cache Initialization File.

To create your cache, call the CacheFactory create function.

The cache object it returns gives access to the native client caching API. For example:

```cpp
CacheFactoryPtr cacheFactory = CacheFactory::createCacheFactory();
CachePtr cachePtr = cacheFactory->create();
```

**Note:** For more information on how to create a cache, see Pivotal GemFire Native Client C++ API, Pivotal GemFire Native Client C# .NET API, and the native client API reference (http://gemfire.docs.pivotal.io/index.html).
Closing the Cache

Use the Cache::close function to release system resources when you finish using the cache.

After the cache is closed, any further method calls on the cache or any region object results in a CacheClosedException.

If the cache is in a durable client, you need to use the keepalive version of the close method. See Disconnecting From the Server.

Cache Initialization File (cache.xml)

To ease the task of managing the structure of the cache, you can define the default Pivotal GemFire cache structure in an XML-based initialization file.

To modify the cache structure, you just edit cache.xml in your preferred text editor. No changes to the application code are required.

This chapter describes the file format of the cache.xml file and discusses its contents.
Cache Initialization File Basics

The contents of the cache initialization file are used to populate or update a cache. This occurs when a cache server starts up, when a client application explicitly creates its cache, or when a client explicitly loads a new structure into an existing cache.

The initialization file can have any name, but is generally referred to as cache.xml. Both client applications and cache servers can use an optional cache.xml file to ease the initialization process.

File Contents

The contents of a declarative XML file correspond to APIs declared in the Cache.hpp and Region.hpp header files. The cache initialization file allows you to accomplish declaratively many of the cache management activities that you can program through the API.

- The contents of the cache initialization file must conform to the XML definition in productDir/dtd/gfccp-cache8000.dtd (see Native Client Cache XML DTD).
- The name of the declarative XML file is specified when establishing a connection to the distributed system. You can define it by setting the cache-xml-file configuration attribute in the gfccp.properties file for the native client. For details about the gfccp.properties file, see Setting System and Cache Properties.
Example cache.xml File

An example cache.xml file shows cache and region initialization for a client. The example shows only a subset of the possible data configurations.

For detailed information about cache and region configuration, including the default attribute settings, see The Native Client Cache. Also see the online API documentation for Cache and RegionAttributes.

For information a cache with server pool, see Using Connection Pools. The example below shows a cache.xml file that creates two regions.

- Region region1 is defined with a full set of region attributes and application plug-ins. The region's entries have RegionTimeToLive and RegionIdleTimeout expiration attributes set (Expiration Attributes).
- Region region2 uses mostly default values.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Sample cache.xml file -->
<!-- Example Declarative Cache Initialization with cache.xml -->
<cache>
<pool name="examplePool" subscription-enabled="true">
  <server host="localhost" port="24680" />
</pool>
<region name="root1" refid="CACHING_PROXY">
  <region-attributes pool-name="examplePool" initial-capacity="25" load-factor="0.32" concurrency-level="10" lru-entries-limit="35">
    <region-idle-time>
      <expiration-attributes timeout="20" action="destroy"/>
    </region-idle-time>
    <entry-idle-time>
      <expiration-attributes timeout="10" action="invalidate"/>
    </entry-idle-time>
    <region-time-to-live>
      <expiration-attributes timeout="5" action="local-destroy"/>
    </region-time-to-live>
    <entry-time-to-live>
      <expiration-attributes timeout="10" action="local-invalidate"/>
    </entry-time-to-live>
  </region-attributes>
</region>
</cache>
```

For details about the individual region attributes, see Region Attributes.
Native Client Cache XML DTD

The contents of the cache initialization file must conform to the data type definitions in gfcpp-cache8000.dtd.

The gfcpp-cache8000.dtd file is in the \texttt{productDir/dtd} directory of your native client installation.

The following example presents the file gfcpp-cache8000.dtd, which defines the XML used by the GemFire native client for declarative caching. The DTD file identifies the valid element tags that may be present in your XML file, the attributes that correspond to each element, and the valid values for the elements and attributes.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE client-cache [ 
<!ELEMENT client-cache (pool*, root-region*, region*, pdx?)>
<!ATTLIST client-cache 
  endpoints CDATA #IMPLIED 
  redundancy-level CDATA #IMPLIED >
<!ELEMENT locator EMPTY>
<!ATTLIST locator 
  host  CDATA #REQUIRED 
  port  CDATA #REQUIRED >
<!ELEMENT server EMPTY>
<!ATTLIST server 
  host  CDATA #REQUIRED 
  port  CDATA #REQUIRED >
<!ELEMENT pdx EMPTY>
<!ATTLIST pdx 
  ignore-unread-fields (false | true) #IMPLIED 
  read-serialized (false | true) #IMPLIED >
<!--
This is the XML DTD for the GemFire Native Client declarative caching XML file.

The contents of a declarative XML file correspond to APIs found in the Gemfire Native Client product, more specifically in the Cache.hpp and Region.hpp files in the product include directory.

A declarative caching XML file is used to populate a Cache when it is created.
-->

<!--
The "client-cache" element is the root element of the declarative cache file.
This element configures a GemFire Native Client Cache and describes the root regions it contains, if any.
-->

<!ELEMENT client-cache (pool*, root-region*, region*, pdx?)>
<!ATTLIST client-cache 
  endpoints CDATA #IMPLIED 
  redundancy-level CDATA #IMPLIED >

<!--
A "locator" element specifies the host and port that a server locator is listening on
-->
<!ELEMENT locator EMPTY>
<!ATTLIST locator 
  host  CDATA #REQUIRED 
  port  CDATA #REQUIRED >

<!--
A "server" element specifies the host and port that a cache server is listening on
-->
<!ELEMENT server EMPTY>
<!ATTLIST server 
  host  CDATA #REQUIRED 
  port  CDATA #REQUIRED >

<!--
A "pdx" element specifies the properties related to pdx types
-->
<!ELEMENT pdx EMPTY>
<!ATTLIST pdx 
  ignore-unread-fields (false | true) #IMPLIED 
  read-serialized (false | true) #IMPLIED >

<!--

}}}
A "pool" element specifies a client-server connection pool.

```xml
<!ELEMENT pool (locator+|server+)>  
<!ATTLIST pool
  free-connection-timeout CDATA #IMPLIED
  load-conditioning-interval CDATA #IMPLIED
  min-connections CDATA #IMPLIED
  max-connections CDATA #IMPLIED
  retry-attempts CDATA #IMPLIED
  idle-timeout CDATA #IMPLIED
  ping-interval CDATA #IMPLIED
  name CDATA #REQUIRED
  read-timeout CDATA #IMPLIED
  server-group CDATA #IMPLIED
  socket-buffer-size CDATA #IMPLIED
  subscription-enabled (false | true) #IMPLIED
  subscription-message-tracking-timeout CDATA #IMPLIED
  subscription-ack-interval CDATA #IMPLIED
  subscription-redundancy CDATA #IMPLIED
  statistic-interval CDATA #IMPLIED
  pr-single-hop-enabled (true | false) #IMPLIED
  thread-local-connections (false | true) #IMPLIED
  multiuser-authentication (false | true) #IMPLIED
>
```

A root-region" element describes a root region whose entries and subregions will be stored in memory.

Note that the "name" attribute specifies the simple name of the region; it cannot contain a "/".

```xml
<!ELEMENT root-region (region-attributes?, region*)>  
<!ATTLIST root-region
  name CDATA #REQUIRED
>
```

A "region" element describes a region (and its entries) in GemFire Native Client cache. Note that the "name" attribute specifies the simple name of the region; it cannot contain a "/".

```xml
<!ELEMENT region (region-attributes?, region*)>  
<!ATTLIST region
  name CDATA #REQUIRED
  refid CDATA #IMPLIED
>
```

A "region-attributes" element describes the attributes of a region to be created. For more details see the RegionFactory header in the product include directory.

```xml
<!ELEMENT region-attributes ((region-time-to-live |
  region-idle-time | entry-time-to-live | entry-idle-time |
  partition-resolver |
  cache-loader | cache-listener | cache-writer | persistence-manager)*)>  
<!ATTLIST region-attributes
  caching-enabled (true | TRUE | false | FALSE) #IMPLIED
  cloning-enabled (true | TRUE | false | FALSE) #IMPLIED
  scope (local | distributed-no-ack | distributed-ack ) #IMPLIED
  initial-capacity CDATA #IMPLIED
  load-factor CDATA #IMPLIED
  concurrency-level CDATA #IMPLIED
  lru-entries-limit CDATA #IMPLIED
  disk-policy (none | overflows | persist ) #IMPLIED
```
endpoints CDATA #IMPLIED
client-notification (true | TRUE | false | FALSE) #IMPLIED
pool-name CDATA #IMPLIED
concurrency-checks-enabled (true | TRUE | false | FALSE) #IMPLIED
id CDATA #IMPLIED
refid CDATA #IMPLIED

<!--
A "region-time-to-live" element specifies a Region's time to live
-->

<!ELEMENT region-time-to-live (expiration-attributes)>

<!--
A "region-idle-time" element specifies a Region's idle time
-->

<!ELEMENT region-idle-time (expiration-attributes)>

<!--
A "entry-time-to-live" element specifies a Region's entries' time to live
-->

<!ELEMENT entry-time-to-live (expiration-attributes)>

<!--
A "entry-idle-time" element specifies a Region's entries' idle time
-->

<!ELEMENT entry-idle-time (expiration-attributes)>

<!--
A "properties" element specifies a persistence properties
-->

<!ELEMENT properties (property*)>

<!--
An "expiration-attributes" element describes expiration
-->

<!ELEMENT expiration-attributes EMPTY>
<!ATTLIST expiration-attributes
  timeout CDATA #REQUIRED
  action (invalidate | destroy | local-invalidate | local-destroy) #IMPLIED
>

<!--
A "cache-loader" element describes a region's CacheLoader
-->

<!ELEMENT cache-loader EMPTY >
<!ATTLIST cache-loader
  library-name CDATA #IMPLIED
  library-function-name CDATA #REQUIRED>
Regions

You create cache regions either programmatically or through declarative statements in the `cache.xml` file. Generally, a cache is organized and populated through a combination of the two approaches.

The region is the core building block of the Pivotal GemFire distributed system. All cached data is organized into data regions and you do all of your data puts, gets, and querying activities against them.
A distributed region can be either non-partitioned or a partitioned region. See the *GemFire User's Guide* for detailed descriptions of both non-partitioned and partitioned regions. Region creation is subject to attribute consistency checks. The requirements for consistency between attributes are detailed both in the online API documentation and throughout the discussion of *Region Attributes*.

See the *GemFire User's Guide* for a complete description of consistency checking in Pivotal GemFire.
Declarative Region Creation

Declarative region creation involves placing the region's XML declaration, with the appropriate attribute settings, in the cache.xml file that is loaded at cache creation.

**Note:** Before creating a region, specify region attributes. See Region Attributes.

Regions are placed inside the cache declaration in region elements. For example:

```xml
<cache>
  <pool name="examplePool" subscription-enabled="true"/>
  <server host="localhost" port="40404"/>
  <region name="A" refid="PROXY">
    <region-attributes pool-name="examplePool"/>
  </region>
  <region name="A1">
    <region-attributes refid="PROXY" pool-name="examplePool"/>
  </region>
  <region name="A2" refid="CACHING_PROXY">
    <region-attributes pool-name="examplePool">
      <region-time-to-live>
        <expiration-attributes timeout="120" action="invalidate"/>
      </region-time-to-live>
    </region-attributes>
  </region>
</cache>
```

The cache.xml file contents must conform to the DTD provided in the productDir/dtd/gfcpp-cache8000.dtd file. For details, see Cache Initialization File.

**Note:** For more information on how to create a region, see Pivotal GemFire Native Client C++ API, Pivotal GemFire Native Client C# .NET API, and the native client API reference (http://gemfire.docs.pivotal.io/index.html).
Programmatic Region Creation

You create regions programmatically with the `regionFactory` class.

**Note:** Before creating a region, specify region attributes. See *Region Attributes.*

Create your regions using the `regionFactory` class.

**C++ RegionFactory Example**

```cpp
RegionFactoryPtr regionFactory = 
    cachePtr->createRegionFactory(CACHING_PROXY);
RegionPtr regPtr0 = regionFactory->setLruEntriesLimit(20000) 
    ->create("exampleRegion0");
```

**Note:** For more information on how to create a region, see *Pivotal GemFire Native Client C++ API*, *Pivotal GemFire Native Client C# .NET API*, and the native client API reference ([http://gemfire.docs.pivotal.io/index.html](http://gemfire.docs.pivotal.io/index.html)).
Invalidating and Destroying Regions

Invalidation marks all entries contained in the region as invalid (with null values). Destruction removes the region and all of its contents from the cache.

You can execute these operations explicitly in the local cache in the following ways:

- Through direct API calls from the native client.
- Through expiration activities based on the region’s statistics and attribute settings.

In either case, you can perform invalidation and destruction as a local or a distributed operation.

- A local operation affects the region only in the local cache.
- A distributed operation works first on the region in the local cache and then distributes the operation to all other caches where the region is defined. This is the proper choice when the region is no longer needed, or valid, for any application in the distributed system.
- If the region on the server is configured as a partitioned region, it cannot be cleared using API calls from the native client.

A user-defined cache writer can abort a region destroy operation. Cache writers are synchronous listeners with the ability to abort operations. If a cache writer is defined for the region anywhere in the distributed system, it is invoked before the region is explicitly destroyed.

Region invalidation and destruction can cause other user-defined application plug-ins to be invoked as well. These plug-ins are described in detail in Application Plug-Ins.

Whether carried out explicitly or through expiration activities, invalidation and destruction cause event notification.
Region Access

You can use `Cache::getRegion` to retrieve a reference to a specified region.

`RegionPtr` returns `NULL` if the region is not already present in the application's cache. A server region must already exist.

A region name *cannot* contain these characters:

- `<`
- `>`
- `:`
- `\`
- `/`
- `|`
- `?`
- `*`
Getting the Region Size

The Region API provides a size method (Size property for .NET) that gets the size of a region.
For native client regions, this gives the number of entries in the local cache, not on the servers.
See the Region online API documentation for details

Region Entries

Region entries hold cached application data. Entries are automatically managed according to region attribute settings.
You can create, update, invalidate, and destroy entries through explicit API calls or through operations distributed from other caches.
When the number of entries is very large, a partitioned region can provide the required data management capacity if the total size of the data is greater than the heap in any single JVM. See the Pivotal GemFire User's Guide for a detailed description of partitioned regions and related data management.
When an entry is created, a new object is instantiated in the region containing:

• The entry key.
• The entry value. This is the application data object. The entry value may be set to NULL, which is the equivalent of an invalid value.

Entry operations invoke callbacks to user-defined application plug-ins. In this chapter, the calls that may affect the entry operation itself (by providing a value or aborting the operation, for example) are highlighted, but all possible interactions are not listed. For details, see Application Plug-Ins.

DateTime objects must be stored in the cache in UTC, so that times correspond between client and server. If you use a date with a different time zone, convert it when storing into and retrieving from the cache.
Entry Distribution Requirements

Entry data distributed among members of the distributed system must be serializable. Entry keys and values are serialized for distribution.

If a native client defines a region, it must register any serializable types for all classes of objects stored in the region. This includes entries that the application gets or puts, as well as entries that are pushed to the client's cache automatically through distribution. The types must be registered before the native client connects to the distributed system.

See Serializing Data for more information about these requirements.
Registering Interest for Entries

For native client regions, you can programmatically register interest in entry keys stored on a cache server region. A client region receives update notifications from the cache server for the keys of interest.

You can register interest for specific entry keys or for all keys. Regular expressions can be used to register interest for keys whose strings match the expression. You can also unregister interest for specific keys, groups of keys based on regular expressions, or for all keys.

**Note:** Interest registration and unregistration are symmetrical operations. Consequently, you cannot register interest in all keys and then unregister interest in a specific set of keys. Also, if you first register interest in specific keys with `registerKeys`, then call `registerAllKeys`, you must call `unregisterAllKeys` before specifying interest in specific keys again.

- **Client API for Registering Interest**
- **Setting Up Client Notification**
- **Registering Interest for Specific Keys**
- **Registering Interest for All Keys**
- **Registering Interest Using Regular Expressions**
- **Register Interest Scenario**

**Client API for Registering Interest**

You register client interest through the C++ or NET API. The C++ API provides the `registerKeys`, `registerAllKeys`, and `registerRegex` methods, with corresponding unregistration accomplished using the `unregisterKeys`, `unregisterAllKeys`, and `unregisterRegex` methods. The .NET API provides the `RegisterKeys`, `RegisterAllKeys`, and `RegisterRegex` methods, with corresponding unregistration accomplished using the `UnregisterKeys`, `UnregisterAllKeys`, and `UnregisterRegex` methods.

The `registerKeys`, `registerRegex` and `registerAllKeys` methods have the option to populate the cache with the registration results from the server. The `registerRegex` and `registerAllKeys` methods can also optionally return the current list of keys registered on the server.

**Setting Up Client Notification**

In addition to the programmatic function calls, to register interest for a server region and receive updated entries you need to configure the region with the `PROXY` or `CACHING_PROXY` `RegionShortcut` setting. The region’s pool should have `subscription-enabled=true` set either in the client XML or programmatically via a `CacheFactory::setSubscriptionEnabled(true)` API call. Otherwise, when you register interest, you will get an `UnsupportedOperationException`.

```xml
<region name = "listenerWriterLoader" refid="CACHING_PROXY">
  ...
</region>
```

Both native clients and Java clients that have subscriptions enabled track and drop (ignore) any duplicate notifications received. To reduce resource usage, a client expires tracked sources for which new notifications have not been received for a configurable amount of time.

**Notification Sequence**

Notifications invoke `CacheListeners` of cacheless clients in all cases for keys that have been registered on the server. Similarly, invalidates received from the server invoke `CacheListeners` of cacheless clients.

If you register to receive notifications, listener callbacks are invoked irrespective of whether the key is in the client cache when a `destroy` or `invalidate` event is received.
Registering Interest for Specific Keys

You register and unregister interest for specific keys through the `registerKeys` and `unregisterKeys` functions. You register interest in a key or set of keys by specifying the key name using the programmatic syntax shown in the following example:

```cpp
keys0.push_back(keyPtr1);
keys1.push_back(keyPtr3);
regPtr0->registerKeys(keys0);
regPtr1->registerKeys(keys1);
```

The programmatic code snippet in the next example shows how to unregister interest in specific keys:

```cpp
regPtr0->unregisterKeys(keys0);
regPtr1->unregisterKeys(keys1);
```

Registering Interest for All Keys

If the client registers interest in all keys, the server provides notifications for all updates to all keys in the region. The next example shows how to register interest in all keys:

```cpp
regPtr0->registerAllKeys();
regPtr1->registerAllKeys();
```

The following example shows a code sample for unregistering interest in all keys.

```cpp
regPtr0->unregisterAllKeys();
regPtr1->unregisterAllKeys();
```

Registering Interest Using Regular Expressions

The `registerRegex` function registers interest in a regular expression pattern. The server automatically sends the client changes for entries whose keys match the specified pattern.

*Keys must be strings in order to register interest using regular expressions.*

The following example shows interest registration for all keys whose first four characters are `Key-`, followed by any string of characters. The characters `.*` represent a wildcard that matches any string.

```cpp
regPtr1->registerRegex("Key-.*");
```

To unregister interest using regular expressions, you use the `unregisterRegex` function. The next example shows how to unregister interest in all keys whose first four characters are `Key-`, followed by any string (represented by the `.*` wildcard).

```cpp
regPtr1->unregisterRegex("Key-.*");
```

Register Interest Scenario

In this register interest scenario, a cache listener is used with a cacheless region that has `subscription-enabled` set to `true`. The client region is configured with caching disabled; client notification is enabled; and a cache listener is established. The client has not registered interest in any keys.

When a value changes in another client, it sends the event to the server. The server will not send the event to the cacheless client, even though `client-notification` is set to `true`.

To activate the cache listener so the cacheless region receives updates, the client should explicitly register interest in some or all keys by using one of the API calls for registering interest. This way, the client receives all events for the keys to which it has registered interest. This applies to Java-based clients as well as native clients.
Using serverKeys to Retrieve a Set of Region Keys

You can retrieve the set of keys defined in the cache server process that are associated with the client region by using the `Region::serverKeys` API function. If the server region is defined as a replicate, the keys returned consist of the entire set of keys for the region.

The following example shows how the client can programmatically call `serverKeys`.

```cpp
VectorOfCacheableKey keysVec;
region->serverKeys( keysVec );
size_t vlen = keysVec.size();
bool foundKey1 = false;
bool foundKey2 = false;
for( size_t i = 0; i < vlen; i++ ) {
    CacheableStringPtr strPtr = dynCast<CacheableStringPtr< keysVec.at( i );
    std::string veckey = strPtr->asChar();
    if ( veckey == "skey1" ) {
        printf( "found skey1" );
        foundKey1 = true;
    }
    if ( veckey == "skey2" ) {
        printf( "found skey2" );
        foundKey2 = true;
    }
}
```

An `UnsupportedOperationException` occurs if the client region is not a native client region. A `MessageException` occurs if the message received from the server could not be handled, which can occur if an unregistered `typeId` is received in the reply.
Adding Entries to the Cache

A region is populated with cached entries in several ways:

- Explicitly, when an application executes a `create` or a `put` operation for a single entry or for multiple entries that do not already exist in the cache.
- Implicitly, when a client does a get on a single entry or on multiple entries that do not already exist in the cache. In this case, the entry is retrieved from a remote cache or through a cache loader (Under Specifying Application Plug-In Attributes see "CacheLoader"). A client can also use `getAll` to populate a region with all values for an array of keys. See Accessing Entries.
- Automatically, when entries are created in remote caches.

If any cache writer is available in the distributed region, it is called before the entry is created and it can abort the creation process.

**Note:** For more information on how to create entries, see Pivotal GemFire Native Client C++ API, Pivotal GemFire Native Client C# .NET API, and the native client API reference (http://gemfire.docs.pivotal.io/index.html).

Adding Entries to the Local Cache

If just the local cache is to be populated, you can either `create` an entry using the `localCreate` Region API, or `put` an entry using `localPut`. See the C++ and .NET online API documentation for details about `localCreate` and `localPut`.

**DateTime** objects must be stored in the cache in UTC, so that times correspond between client and server. If you use a date with a different time zone, convert it when storing into and retrieving from the cache. This example converts a local time to UTC for a put operation:

```java
DateTime t1( 2009, 8, 13, 4, 11, 0, DateTimeKind.Local);
region0.Put( 1, t1.ToUniversalTime() );
```

Adding Multiple Entries Using PutAll

If you need to add many cache entries to a region at one time, you can improve cache performance by using the `putAll` function to add them in a single distributed operation. Multiple key/value pairs are stored in a hashmap, then the hashmap contents are processed on the server as either new entries, updates, or invalidates for existing entries.

Under Adding an Entry to the Cache see "Bulk Put Operations Using putAll" for more information about the `putAll` API. Additional details are available in the online API documentation for `Region::putAll` (C++), or `Region.PutAll` (.NET).
Updating Entries

A cached entry can be updated using these methods:

- Explicitly, when a client invokes a `put` operation on an existing entry.
- Implicitly, when a `get` is performed on an entry that has an invalid value in the cache. An entry can become invalid through an explicit API call, through an automated expiration action, or by being created with a value of null.
- Automatically, when a new entry value is distributed from another cache.

Similar to entry creation, all of these operations can be aborted by a cache writer.

The `get` function returns a direct reference to the entry value object. A change made using that reference is called an in-place change because it directly modifies the contents of the value in the local cache. For details on safe cache access, see *Managing the Lifetime of a Cached Object.*
Accessing Entries

Use the API to retrieve the entry key, entry value, and the RegionEntry object itself. A variety of functions provide information for individual entries and for the set of all entries resident in the region. The online API documentation lists all available access functions.

A region's entry keys and RegionEntry objects are directly available from the local cache. Applications can directly access the local cache's stored entry value through the RegionEntry::getValue function. The getValue function either returns the value if a valid value is present in the local cache, or NULL if the value is not valid locally. This function does no data loading, nor does it look elsewhere in the distributed system for a valid value.

**Note:** Direct access through RegionEntry::getValue does not reset an entry's timestamp for LRU expiration. See Specifying Expiration Attributes for more information about LRU expiration.

In comparison, the standard Region::get functions consider all caches and all applicable loaders in the distributed system in an attempt to return a valid entry value to the calling application. The primary attribute setting affecting entry retrieval is CacheLoader. See Specifying Application Plug-In Attributes.

The standard Region::get functions may implement a number of operations in order to retrieve a valid entry value. The operations used depend on the region's attribute settings and on the state of the entry itself. By default, the client retrieves entry values through calls to the get function. The client can override this behavior for any region by defining a cache loader for the region.

The following sections discuss the get function and special considerations for entry retrieval.

Entry Retrieval

You retrieve entry values with the Region::get function.

When an entry value is requested from a region, it is either retrieved from the cache server or fetched by the region's locally-defined cache loader in this sequence:

1. local cache search
2. server cache
3. local load (For distributed regions, the local load is fetched before remote cache values)

How the get Operation Affects the Local Entry Value

If a get operation retrieves an entry value from outside the local cache through a local load, it automatically puts the value into the cache for future reference.

Note that these load operations do not invoke a cache writer. Because the loader and writer operate against the same data source, you do not need to perform a cache write for entries that were just fetched from that data source. For descriptions of these processes, see Specifying Application Plug-In Attributes.

**Note:** Access through a get operation resets an entry's timestamp for LRU expiration.

Getting Multiple Entries Using getAll

You can use the Region API to get all values for an array of keys from the local cache or cache server. Under Bulk Put Operations Using putAll for more information.
Invalidating or Destroying Cached Entries

Invalidating an entry sets the entry's value to NULL. Destroying it removes the entry from the region altogether. These operations can be carried out in the local cache in the following ways:

- Through direct API calls from the client.
- Through expiration activities based on the entry's statistics and the region's attribute settings.

**Note:** A user-defined cache writer is called before an operation is completed, and can abort an entry destroy operation.

Whether carried out explicitly or through expiration activities, invalidation and destruction cause event notification: The CacheEvent object has an isExpiration flag that is set to true for events resulting from expiration activities, and set to false for all other events.
Notification for Operations

Listeners are invoked for client-initiated operations only after the client operation succeeds on the server. Listener invocation on the client indicates that the server has the same data as the client.

If a client operation fails on the server, the operation is rolled back, assuming that no other thread has modified the data in the intervening period. Rollback may not be possible in cases where the entry has been evicted by LRU or expiration during this period. Thus when an exception is received from the server for an operation, local changes may not have been rolled back.

Event Notification Sequence

Events received on the clients that originated on the server invoke the subscription for the event as seen by the server. Events originating on the client invoke the subscription as seen by the client.

For example, a client that receives a `create` and an `update` from the server fires a `create` event and an `update` event because that is how the server saw it. A cacheless client that does two consecutive `put` operations has two `afterCreate` events invoked on the originating client because the client does not have any history about the first `put`, since it is cacheless.

For the same sequence, the server sees an `afterCreate` and an `afterUpdate` event, and a remote client receiving the event sees an `afterCreate` followed by an `afterUpdate` event. A client that caches locally sees an `afterCreate` and an `afterUpdate` for the same scenario (as will the server and remote clients).

Region Consistency

GemFire ensures that all copies of a region eventually reach a consistent state on all members and clients that host the region.

By default GemFire members perform consistency checks when they apply updates to a distributed region, in order to ensure that all copies of the region eventually become consistent on all GemFire members and client caches that host the region. Different types of region ensure consistency using different techniques. However, when consistency checking is enabled (the default) all entries in a region require additional overhead in order to store version and timestamp information.

Although a region must have the same consistency checking configuration on all GemFire members that host the region, you can optionally disable consistency checking in a client cache region while leaving consistency checking enabled for the region on GemFire members. This configuration may be necessary in certain cases where the client must view all updates to a given region, even when GemFire members discard an update in order to preserve region consistency.

Native Client Overhead for Consistency Checks

In the native client regions, the overhead for performing consistency check is an additional 11 bytes per region entry. This overhead is slightly smaller than the overhead required to provide consistency checking on server-side region entries.

If you cannot support the additional overhead in your deployment, you can disable consistency checks by setting the region attribute `concurrency-checks-enabled` to "false" for each region hosted by your client.
Region Attributes

Region attributes govern the automated management of a region and its entries.
Region attribute settings determine where the data resides, how the region is managed in memory, and the automatic loading, distribution, and expiration of region entries.
Specifying Region Attributes

Region attributes govern the automated management of a region and its entries.

Specify region attributes before creating the region. You can do this either through the declarative XML file or through the API. The API includes classes for defining a region's attributes before creation and for modifying some of them after creation. For details, see the online API documentation for `RegionShortcut`, `RegionAttributes`, `AttributesFactory`, and `AttributesMutator`. 
RegionShortcuts

GemFire provides a number of predefined, shortcut region attributes settings for your use, in RegionShortcut.

Shortcut attributes are a convenience only. They are named attributes that GemFire has already stored for you. You can override their settings by storing new attributes with the same id as the predefined attributes.

You can also create custom region attributes and store them with an identifier for later retrieval. Both types of stored attributes are referred to as named region attributes. You can create and store your attribute settings in the cache.xml file and through the API.

Retrieve named attributes by providing the ID to the region creation. This example uses the shortcut CACHING_PROXY attributes to create a region:

```xml
<region name="testRegion" refid="CACHING_PROXY"/>
```

You can modify named attributes as needed. For example, this adds a cache listener to the region:

```xml
<region name="testRegion" refid="CACHING_PROXY">
    <region-attributes>
        <cache-listener library-name="myAppLib"
            library-function-name="myCacheListener"/>
    </region-attributes>
</region>
```

In this example, the modified region shortcut is saved to the cache using the region attribute id, for retrieval and use by a second region:

```xml
<region name="testRegion" refid="CACHING_PROXY">
    <region-attributes id="Caching_Proxy_With_Listener">
        <cache-listener library-name="myAppLib"
            library-function-name="myCacheListener"/>
    </region-attributes>
</region>
<region name="newTestRegion" refid="Caching_Proxy_With_Listener"/>
```

Shortcut Attribute Options

You can select the most common region attributes settings from RegionShortcut, GemFire's predefined named region attributes.

This section provides an overview of the options available in the region shortcut settings.

**Note:** For complete listings and descriptions, including information on the underlying RegionAttributes settings for each shortcut, see the online documentation for RegionShortcut.

These are the options available in RegionShortcut.

**Communication with Servers and Data Storage**

- **PROXY** does not store data in the client cache, but connects the region to the servers for data requests and updates, interest registrations, and so on.
- **CACHING_PROXY** stores data in the client cache and connects the region to the servers for data requests and updates, interest registrations, and so on.
- **LOCAL** stores data in the client cache and does not connect the region to the servers. This is a client-side-only region.

**Data Eviction**

For the non-PROXY regions--the regions that store data in the client cache--you can add data eviction:
• **ENTRY_LRU** causes least recently used data to be evicted from memory when the region reaches the entry count limit.
Mutable and Immutable Region Attributes

Attributes that are immutable (fixed) after region creation govern storage location, data distribution, statistics, application plug-ins, and the configuration and management of the region's data hashmap. This table lists the immutable attributes and their default settings.

Table 12: Immutable Region Attributes

<table>
<thead>
<tr>
<th>Immutable Region Attribute</th>
<th>Default Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>See CachingEnabled</td>
<td>true</td>
</tr>
<tr>
<td>See InitialCapacity</td>
<td>16 (entries)</td>
</tr>
<tr>
<td>See LoadFactor</td>
<td>0.75</td>
</tr>
<tr>
<td>See ConcurrencyLevel</td>
<td>16</td>
</tr>
<tr>
<td>See DiskPolicy</td>
<td></td>
</tr>
<tr>
<td>See PersistenceManager</td>
<td>NULL</td>
</tr>
<tr>
<td>PartitionResolver. See Specifying Application Plug-In Attributes.</td>
<td></td>
</tr>
</tbody>
</table>

Mutable region attributes identify expiration and cache listener, cache writer and cache loader actions that are run from the defining client. The next table lists the mutable attributes that generally can be modified after region creation by using the AttributesMutator for the region.

Table 13: Mutable Region Attributes

<table>
<thead>
<tr>
<th>Mutable Region Attribute</th>
<th>Default Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expiration attributes. See Specifying Expiration Attributes.</td>
<td>no expiration</td>
</tr>
<tr>
<td>See LruEntriesLimit.</td>
<td>0 (no limit)</td>
</tr>
<tr>
<td>CacheLoader. See Specifying Application Plug-In Attributes.</td>
<td></td>
</tr>
<tr>
<td>CacheWriter. See Specifying Application Plug-In Attributes.</td>
<td></td>
</tr>
<tr>
<td>CacheListener. See Specifying Application Plug-In Attributes.</td>
<td></td>
</tr>
</tbody>
</table>

Under Specifying Application Plug-In Attributes, see "Using AttributesMutator to Modify a Plug-in" for information about using AttributesMutator with cache listeners, cache loaders, and cache writers.

The remainder of this chapter examines these attributes in detail. Throughout the descriptions, cache.xml file snippets show how each attribute can be set declaratively.
CachingEnabled

This attribute determines whether data is cached in this region. For example, you might choose to configure the distributed system as a simple messaging service where clients run without a cache.

**Note:** You can configure the most common of these options with GemFire’s predefined region attributes. See `RegionShortcuts` and the Javadocs for `RegionShortcut`.

If `CachingEnabled` is false (no caching), an `IllegalStateException` is thrown if any of these attributes are set:

- `InitialCapacity`
- `EntryTimeToLive` under **Specifying Expiration Attributes**
- `EntryIdleTimeout` under **Specifying Expiration Attributes**
- `LoadFactor`
- `ConcurrencyLevel`
- `LruEntriesLimit`
- `DiskPolicy`

The following declaration enables caching for the region:

```
<region-attributes caching-enabled="true"/>
```
InitialCapacity

Use this attribute, together with the LoadFactor attribute, to set the initial parameters on the underlying hashmap that stores region entries. This is the number of entries that the region map will be ready to hold when it is created.

This declaration sets the region's initial capacity to 10000:

```xml
<region-attributes initial-capacity="10000"/>
```
**LoadFactor**

Use this attribute, together with the `InitialCapacity` attribute, to set the initial parameters on the underlying hashmap that stores region entries. When the number of entries in the map exceeds the `LoadFactor` times current capacity, the capacity is increased and the map is rehashed. You get the best performance if you configure a properly sized region at the start and do not have to rehash it.

This declaration sets the region's load factor to 0.75:

```xml
<region-attributes load-factor="0.75"/>
</region-attributes>
```
**ConcurrencyLevel**

This attribute estimates the maximum number of application threads that concurrently access a region entry at one time. This attribute helps optimize the use of system resources and reduce thread contention.

The following declaration sets the region's `ConcurrencyLevel` to 16:

```
<region-attributes concurrency-level="16">
</region-attributes>
```

**Note:** When `CachingEnabled` is false, do not set the `ConcurrencyLevel` attribute. An `IllegalStateException` is thrown if the attribute is set.
**ConcurrencyChecksEnabled**

This attribute determines whether members perform checks to provide consistent handling for concurrent or out-of-order updates to distributed regions.

A client cache can disable consistency checking for a region even if server caches enable consistency checking for the same region. This configuration ensures that the client sees all events for the region, but it does not prevent the client cache region from becoming out-of-sync with the server cache.

Optionally enable concurrency checks for the region. Example:

```xml
<region-attributes concurrency-checks-enabled="true">
</region-attributes>
```

See *Region Consistency* for more information.
LruEntriesLimit

This attribute sets the maximum number of entries to hold in a caching region. When the capacity of the caching region is exceeded, a least-recently-used (LRU) algorithm is used to evict entries.

**Note:** This is a tuning parameter that affects system performance.

When eviction is configured, memory consumption or entry count is monitored and, when capacity is reached, GemFire makes way for new entries by removing or overflowing the stalest LRU entries to disk.

If you use disk data overflow to supplement memory for your data cache, make sure you have enough disk space to store the data.

This declaration limits the region to 20,000 entries:

```xml
<region-attributes lru-entries-limit="20000"
    initial-capacity="20000"
    load-factor="1">
</region-attributes>
```

Evicted entries can be destroyed or moved to disk as an extension of the cache. See [DiskPolicy](#).

**Note:** When `CachingEnabled` is false, do not set the `LruEntriesLimit` attribute. An `IllegalStateException` is thrown if the attribute is set.

See also [Controlling Cache Size](#).
DiskPolicy

If the `lru-entries-limit` attribute is greater than zero, the optional `disk-policy` attribute determines how over-limit LRU entries are handled. LRU entries over the limit are either destroyed by default (`disk-policy` is `none`) or written to disk (`overflows`).

**Note:** If `LruEntriesLimit` is 0, or `CachingEnabled` is false, do not set the `disk-policy` attribute. An `IllegalStateException` is thrown if the attribute is set.

This declaration causes LRU to overflow to disk:

```xml
<region-attributes lru-entries-limit="20000"
    disk-policy="overflows">
    <persistence-manager ... />
</region-attributes>
```

Overflow requires a persistence manager for cache-to-disk and disk-to-cache operations. See `PersistenceManager`.

**Overflowing Data to Disk**

Region data can be stored to disk using the overflow process to satisfy region capacity restrictions without completely destroying the local cache data. The storage mechanism uses disk files to hold region entry data. When an entry is overflowed, its value is written to disk but its key and entry object remain in the cache. This also uses the region attribute `DiskPolicy`.

Overflow allows you to keep the region within a user-specified size in memory by relegating the values of least recently used (LRU) entries to disk. Overflow essentially uses disk as a swap space for entry values. When the region size reaches the specified threshold, entry values are moved from memory to disk, as shown in the following figure. If an entry is requested whose value is only on disk, the value is copied back into memory, possibly causing the value of a different LRU entry to be overflowed to disk.

![Data Flow Between Overflow Region and Disk Files](image)

**Figure 2: Data Flow Between Overflow Region and Disk Files**

In this figure the value of the LRU entry X has been moved to disk to recover space in memory. The key for the entry remains in memory. From the distributed system perspective, the value on disk is as much a part of the region as the data in memory. A `get` performed on region B looks first in memory and then on disk as part of the local cache search.
**PersistenceManager**

For each region, if the disk-policy attribute is set to overflows, a persistence-manager plug-in must perform cache-to-disk and disk-to-cache operations. See *Application Plug-Ins*.

Persistence manager declaration:

```xml
<region-attributes lru-entries-limit="nnnnn"
  disk-policy="overflows">
  <persistence-manager library-name="libraryName"
    library-function-name="functionName">
    <properties>
      <property name="propertyName" value="propertyValue" />
    </properties>
  </persistence-manager>
</region-attributes>
```

The optional properties set parameters for the plug-in.

- **Using SQLite as a Persistence Manager**
- **SQLite Persistence Manager Region Attributes**
- **Configuring the SQLite Persistence Manager Plug-In for C++ Applications**
- **Configuring the SQLite Persistence Manager Plug-In for .NET Applications**
- **Implementing a PersistenceManager with the IPersistenceManager Interface**

**Using SQLite as a Persistence Manager**

The GemFire native client distribution includes a persistence manager that uses the open-source SQLite library.

SQLite is a software library that implements a self-contained transactional SQL database. SQLite does not require its own server or separate configuration, and the source code for SQLite is in the public domain. For more information on SQLite, see [http://www.sqlite.org](http://www.sqlite.org).

Each SQLite persistence manager persists its region data in a SQLite database that is stored in disk files. In a given native client application process, each region must have a unique persistence (overflow) directory.

![SQLite Database Persistence Manager Directory Structure](image-url)

**SQLite Persistence Manager Region Attributes**

The following table describes the region attributes that can be configured for the SQLite persistence manager.
**Configuring the Native Client Cache**

**Pivotal GemFire Native Client**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Default Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>PersistenceDirectory</td>
<td>Directory where each region's database files are stored. This setting must be different for each region including regions in different processes. This directory is created by the persistence manager. The persistence manager fails to initialize if this directory already exists or cannot be created.</td>
<td>Default is to create a subdirectory named GemFireRegionData in the directory where the process using the region was started.</td>
</tr>
<tr>
<td>PageSize</td>
<td>Maximum page size of the SQLite database. SQLite can limit the size of a database file to prevent the database file from growing too large and consuming too much disk space.</td>
<td>Ordinarily, if no value is explicitly provided, SQLite creates a database with the page size set to SQLITE_DEFAULT_PAGE_SIZE (default is 1024). However, based on certain device characteristics (for example, sector-size and atomic write() support) SQLite may choose a larger value. PageSize specifies the maximum value that SQLite will be able to choose on its own. See <a href="http://www.sqlite.org/compile.html#default_page_size">http://www.sqlite.org/compile.html#default_page_size</a> for more details on SQLITE_DEFAULT_PAGE_SIZE.</td>
</tr>
<tr>
<td>MaxPageCount</td>
<td>Maximum number of pages in one database file.</td>
<td>SQLite default, which is 1073741823.</td>
</tr>
</tbody>
</table>

**Configuring the SQLite Persistence Manager Plug-In for C++ Applications**

To load the SQLite persistence manager plug-in for C++ applications, you can configure it either in your client's cache.xml or programmatically using the native client C++ API.

The following is an example of how to specify the following region attributes in your client's cache.xml:

```xml
<region-attributes>
  <persistence-manager library-name="libSqLiteImpl.so" library-function-name="createSqLiteInstance">
    <properties>
      <property name="PersistenceDirectory" value="/xyz"/>
      <property name="PageSize" value="65536"/>
      <property name="MaxPageCount" value="1073741823"/>
    </properties>
  </persistence-manager>
</region-attributes>
```

**C++ API Example**

To use the native client C++ API, set SQLite persistence manager attributes programmatically as follows:

```cpp
PropertiesPtr sqliteProperties = Properties::create();
sqliteProperties->insert("MaxPagecount","5");
sqliteProperties->insert("PageSize","1024");
sqliteProperties->insert("PersistenceDirectory","Sqlite-Test779");
```
Configuring the SQLite Persistence Manager Plug-In for .NET Applications

To load the SQLite persistence manager plug-in for .NET applications, you can configure it either in your client's cache.xml or programmatically using the .NET API:

```xml
<persistence-manager library-name="Gemstone.Gemfire.Plugins.Sqlite"
  <properties>
    <property name="PersistenceDirectory" value="Sqlite"/>
    <property name="MaxPageCount" value="1073741823"/>
    <property name="PageSize" value="65536"/>
  </properties>
</persistence-manager>
```

.NET API Example

To use the native client .NET API, set the SQLite persistence manager attributes programmatically as follows:

```csharp
Properties<string, string> sqliteProperties = new Properties<string, string>();
sqliteProperties.Insert("PageSize", "65536");
sqliteProperties.Insert("MaxFileSize", "51200000");
sqliteProperties.Insert("PersistenceDirectory", SqliteDir);
    sqliteProperties);
```

You can also use and configure the C++ SQLite persistence manager library from your .NET application as follows:

```csharp
rf.SetPersistenceManager("SqliteImpl", "createSqliteInstance",
    sqliteProperties);
```

Implementing a PersistenceManager with the IPersistenceManager Interface

When developing .NET managed applications, you can use the IPersistenceManager managed interface to implement your own persistence manager. The following code sample provides the IPersistenceManager interface:

```csharp
/// <summary>
/// IPersistenceManager interface for persistence and overflow.
/// This class abstracts the disk-related operations in case of persistence or overflow to disk.
/// A specific disk storage implementation will implement all the methods described here.
/// </summary>
generic<class TKey, class TValue>
public interface class IPersistenceManager
{
    public:
        /// <summary>
        /// Called after an implementation object is created. Initializes all the implementation specific environments needed.
        /// </summary>
```
/// <summary>
/// Region for which this PersistenceManager is initialized.
/// </summary>
/// <param name="region">
/// Region for which this PersistenceManager is initialized.
/// </param>
/// <param name="diskProperties">
/// Configuration Properties used by PersistenceManager implementation.
/// </param>
void Init(IRegion<TKey, TValue>^ region, Properties<String^, String^>^ diskProperties);

/// <summary>
/// Writes a key, value pair of region to the disk. The actual file or database related write operations should be implemented in this method.
/// </summary>
/// <param name="key">
/// the key to write.
/// </param>
/// <param name="value">
/// the value to write.
/// </param>
void Write(TKey key, TValue value);

/// <summary>
/// This method is not used.
/// </summary>
bool WriteAll();

/// <summary>
/// Reads the value for the key from the disk.
/// </summary>
/// <param name="key">
/// key for which the value has to be read.
/// </param>
TValue Read(TKey key);

/// <summary>
/// This method is not used.
/// </summary>
bool ReadAll();

/// <summary>
/// Destroys the entry specified by the key in the argument.
/// </summary>
/// <param name="key">
/// key of the entry which is being destroyed.
/// </param>
void Destroy(TKey key);

/// <summary>
/// Closes the persistence manager instance.
/// </summary>
void Close();

The following is a sample interface implementation:

class MyPersistenceManager<TKey, TValue> : IPersistenceManager<TKey, TValue>
{
    #region IPersistenceManager<TKey,TValue> Members
    public void Close()
    {
        throw new NotImplementedException();
    }

    public void Destroy(TKey key)
    {
        throw new NotImplementedException();
    }

    #endregion
}
public void Init(IRegion<TKey, TValue> region, Properties<string, string> diskProperties)
{
    throw new NotImplementedException();
}

public TValue Read(TKey key)
{
    throw new NotImplementedException();
}

public void Write(TKey key, TValue value)
{
    throw new NotImplementedException();
}

public bool ReadAll()
{
    throw new NotImplementedException();
}

public bool WriteAll()
{
    throw new NotImplementedException();
}

#endregion
Specifying Application Plug-In Attributes

The plug-in attributes allow you to customize client region behavior for loading, updating, deleting, and overflowing region data and for accessing data in server partitioned regions. All client plug-ins are available through the C++ and .NET API.

Application plug-ins for cache regions in clients can be declared either programmatically or in the cache.xml file.

- **Figure 4: Where Application Plug-Ins Run**
- **Overview of Application Plug-ins**
- **CacheLoader**
- **CacheWriter**
- **CacheListener**
- **PartitionResolver**
- **Using AttributesMutator to Modify a Plug-In**
- **Considerations for Implementing Callbacks**

**Figure 4: Where Application Plug-Ins Run**

**Overview of Application Plug-ins**

The API provides the framework for application plug-ins with callback functions for the appropriate events. Your classes and functions can customize these for your application's needs. When creating a region, specify these as part of the region's attributes settings. For regions already in the cache, you can specify new CacheLoader, CacheWriter, and CacheListener using the region's AttributeMutator. The PartitionResolver is not mutable.

- **CacheLoader**: A data loader called when an entry get operation fails to find a value for a given key. A cache loader is generally used to retrieve data from an outside source such as a database, but it may perform any operation defined by the user. Loaders are invoked as part of the distributed loading activities for entry retrieval, described in *Entry Retrieval*.

- **CacheWriter**: A synchronous event listener that receives callbacks before region events occur and has the ability to abort the operations. Writers are generally used to keep a back-end data source synchronized with the cache.

- **CacheListener**: An asynchronous event listener for region events in the local cache.

- **PartitionResolver**: Used for single-hop access to partitioned region entries on the server side. This resolver implementation must match that of the PartitionResolver on the server side.

The following XML declaration specifies a cache loader for a region when the region is created.

```
<region-attributes>
  <cache-loader library-name="appl-lib"
                library-function-name ="createCacheLoader">
```
The rest of this section gives more detailed descriptions of these application plug-ins, followed by special considerations for plug-ins in distributed regions and some guidelines for writing callbacks.

**CacheLoader**

A cache loader is an application plug-in used to load data into the region. When an entry is requested that is unavailable in the region, a cache loader may be called upon to load it. Generally, you use a cache loader to retrieve the data from a database or another source outside the distributed system, but it may perform any operation defined by the user.

The `CacheLoader` interface provides one function, `load`, for customizing region entry loading. A distributed region may have cache loaders defined in any or all caches where the region is defined. When loading an entry value, a locally defined cache loader is always used before a remote loader. In distributed regions, loaders are available for remote entry retrieval.

**CacheWriter**

A cache writer is an application plug-in that synchronously handles changes to a region's contents. It is generally used to keep back-end data sources synchronized with a cache region. A cache writer has callback functions to handle region destruction and entry creation, update, and destruction. These functions are all called before the modification has taken place and can abort the operation.

You can also use cache writers to store data that you want to make persistent.

**CacheListener**

A cache listener is an application plug-in that asynchronously handles changes to a region's contents. A cache listener has callback functions to handle region destruction and invalidation, along with entry creation, update, invalidation, and destruction. These functions are called asynchronously after the modification has taken place.

This declarative XML example establishes a cache listener when a region is created:

```xml
<region name="region11">
  <region-attributes>
    <cache-listener library-name="appl-lib" library-function-name="createCacheListener" />
  </region-attributes>
</region>
```

Unlike cache loaders and cache writers, cache listeners only receive events for entries to which the client has performed operations or registered interest.

When the listener is attached to a region with caching disabled, the old value is always `NULL`.

**Note:** Do not perform region operations inside the cache listener. Once you have configured a cache listener, the event supplies the new entry values to the application. Performing a get with a key from the `EntryEvent` can result in distributed deadlock. For more about this, see the online API documentation for `EntryEvent`.

When a region disconnects from a cache listener, you can implement the `afterRegionDisconnected` callback event. This callback event is only be invoked when using the pool API and subscription is enabled on the pool. For example:

```c++
class DisconnectCacheListener : public CacheListener
{
  void afterRegionDisconnected( const RegionPtr& region )
  {
    printf("After Region Disconnected event received");
  }
};
```
PartitionResolver

This section pertains to data access in server regions that have custom partitioning. Custom partitioning uses a Java `PartitionResolver` to colocate like data in the same buckets. For the client, you can use a `PartitionResolver` that matches the server's implementation to access data in a single hop. With single-hop data access, the client pool maintains information on where a partitioned region's data is hosted. When accessing a single entry, the client directly contacts the server that hosts the key--in a single hop.

**Note:** Single hop is used for the following operations: `put`, `get`, `destroy`, `putAll`, `getAll`, `removeAll` and `onRegion` function execution.

Implementing Single-Hop on a Partitioned Region

1. Make sure the pool attribute, `pr-single-hop-enabled`, is set to `true` or not set. It is `true` by default.
2. If the server uses a custom `PartitionResolver` install an implementation of `PartitionResolver` in the client region that returns, entry for entry, the same value as the server's Java `PartitionResolver` implementation. The server uses the resolver to colocate like data within a partitioned region.

If the server does not use a custom resolver, the default resolvers in client and server match, so single hop will work there by default.

Disable single hop behavior for a region by setting its pool attribute `pr-single-hop-enabled` to `false`.

See `<pool>` in the GemFire *User's Guide* for information on setting `pr-single-hop-enabled`.

See the GemFire *User's Guide* for more information, including colocating like data within a partitioned region and how to get the best performance with PR single hop.

Implementing a PartitionResolver

See the GemFire *User's Guide* for information on custom-partitioning the server partitioned regions.

1. Implement `PartitionResolver` in the same place that you did in the server--custom class, key, or cache callback argument.
2. Program the resolver's functions the same way you programmed them in the Java implementation. Your implementation must match the server's.

Example of programming the `PartitionResolver` in C++:

```cpp
class TradeKeyResolver : public PartitionResolver
{
private:
    string m_tradeID;
    int m_month;
    int m_year;
public:
    TradeKeyResolver() { }
    TradeKeyResolver(int month, int year) {
        m_month = month;
        m_year = year;
    }
    ~TradeKeyResolver() { }

    static PartitionResolverPtr createTradeKeyResolver() {
        PartitionResolverPtr tradeKeyResolver(new TradeKeyResolver);
        return tradeKeyResolver;
    }
    const char* getName() {
        return "TradeKey";
    }
    CacheableKeyPtr getRoutingObject(const EntryEvent& opDetails) {
        return CacheableKey::create(m_month + m_year);
    }
};
```
Example of programming the `PartitionResolver` in C#:

```csharp
using System;
using System.Threading;
// Use the GemFire namespace

class TradeKeyResolver : IPartitionResolver
{
    private int m_month = 0;
    private int m_year = 0;

    public static TradeKeyResolver CreateTradeKeyResolver()
    {
        return new TradeKeyResolver();
    }

    public virtual ICacheableKey GetRoutingObject(EntryEvent entry)
    {
        return new CacheableInt32(m_month + m_year);
    }

    public virtual String GetName()
    {
        return "TradeKeyResolver";
    }
}
```

3. Install the resolver in the region. Use one of these methods:

**XML partition resolver declaration:**

```xml
<region name="trades" refid="CACHING_PROXY">
    <region-attributes>
        <partition-resolver library-name="appl-lib" library-function-name="createTradeKeyResolver"/>
    </region-attributes>
</region>
```

**Programmatic partition resolver installation:**

```csharp
void setPartitionResolver()
{
    CachePtr cachePtr = CacheFactory::createCacheFactory()->create();
    PartitionResolverPtr resolver( new TradeKeyResolver());
    RegionFactoryPtr regionFactory = cachePtr->createRegionFactory(PROXY)
        ->setClientNotificationEnabled(true)
        ->setPartitionResolver(resolver);
    RegionPtr regionPtr = regionFactory->create( "Trades" );
}
```

Your implementation of `PartitionResolver` must match that of the server side.
Using AttributesMutator to Modify a Plug-In

A cache listener, cache loader or cache writer can be added to or removed from a region after the region is created by retrieving and running the Region object's AttributesMutator. Mutable attributes define operations that are run from the client itself.

This example shows how to use AttributesMutator to dynamically add a cache listener to an existing region.

```cpp
void setListener(RegionPtr& region)
{
    CacheListenerPtr regionListener = new TestCacheListener();
    AttributesMutatorPtr regionAttributesMutator = region->getAttributesMutator();
    // Change cache listener for region.
    regionAttributesMutator->setCacheListener(regionListener);
}
```

The plug-ins can also be implemented using a dynamically linked library. The class is not available to the application code in this case, so a factory method is required by the set function along with the name of the library.

This example shows how to use AttributesMutator along with the setCacheListener function to obtain a new cache listener object using the factory function provided by the library. Next, the listener is set for the region.

```cpp
void setListenerUsingFactory(RegionPtr& region)
{
    AttributesMutatorPtr regionAttributesMutator = region->getAttributesMutator();
    // Change cache listener for region.
    regionAttributesMutator->setCacheListener("Library", "createTestCacheListener");
}
```

To use AttributesMutator to remove a plug-in from a region, set the plug-in's value to NULLPTR, as shown in the following example.

```cpp
void removeListener(RegionPtr& region)
{
    CacheListenerPtr nullListener = NULLPTR;
    AttributesMutatorPtr regionAttributesMutator = region->getAttributesMutator();
    // Change cache listener for region to NULLPTR
    regionAttributesMutator->setCacheListener(nullListener);
}
```

Considerations for Implementing Callbacks

Keep your callback implementations lightweight and prevent situations that might cause them to hang. For example, do not perform distribution operations or disconnects inside events.

Your code should handle any exceptions that it generates. If not, GemFire handles them as well as possible. Because C++ has no standard for exceptions, in many cases GemFire can only print an unknown error message.
Specifying Expiration Attributes

Expiration attributes govern the automatic eviction of regions and region entries from the cache. Eviction is based on the time elapsed since the last update or access to the object. This is referred to as the least-recently-used (LRU) eviction process. Expiration options range from marking the expired object as invalid to completely removing it from the distributed cache. Eviction can help keep data current by removing outdated entries, prompting a reload the next time they are requested. Eviction may also be used to recover space in the cache by clearing out unaccessed entries and regions.

Similar to application plug-ins, expiration activities are hosted by each application that defines a region in its cache.

The following example shows a declaration that causes the region's entries to be invalidated in the local cache after they have not been accessed for one minute.

```
<region-attributes>
  <entry-idle-time>
    <expiration-attributes timeout="60" action="local-invalidate"/>
  </entry-idle-time>
</region-attributes>
```

Region and region entry expiration attributes are set at the region level. By default, regions and entries do not expire. The following attributes cover two types of expiration: time-to-live (TTL) and idle timeout.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegionTimeToLive</td>
<td>Number of seconds that the region remains in the cache after the last creation or update before the expiration action occurs.</td>
</tr>
<tr>
<td>EntryTimeToLive</td>
<td>For entries, the counter is set to zero for create and put operations. Region counters are reset when the region is created and when an entry has its counter reset. An update to an entry causes the time-to-live (TTL) counters to be reset for the entry and its region.</td>
</tr>
<tr>
<td>RegionIdleTimeout</td>
<td>Number of seconds that the region remains in the cache after the last access before the expiration action occurs.</td>
</tr>
<tr>
<td>EntryIdleTimeout</td>
<td>The idle timeout counter for an object is reset when its TTL counter is reset. An entry's idle timeout counter is also reset whenever the entry is accessed through a get operation. The idle timeout counter for a region is reset whenever the idle timeout is reset for one of its entries.</td>
</tr>
</tbody>
</table>

Using Statistics to Measure Expiration

Expiration is measured by comparing expiration attribute settings with the last accessed time and last modified time statistics. These statistics are directly available to applications through the CacheStatistics object that is returned by the Region::getStatistics and RegionEntry::getStatistics functions. The CacheStatistics object also provides a function for resetting the statistics counters.

Expiration Actions

You can specify one of the following actions for each expiration attribute:
• **Destroy.** Removes the object completely from the cache. For regions, all entries are destroyed as well. Destroy actions are distributed according to the region's distribution settings.

• **Invalidate.** Marks the object as invalid. For entries, the value is set to **null**. You invalidate a region by invalidating all its entries. Invalidate actions are distributed according to the region's distribution settings. When an entry is invalid, a **get** causes the cache to retrieve the entry according to the steps described in *Entry Retrieval*.

• **Local destroy.** Destroys the object in the local cache but does not distribute the operation.

• **Local invalidate.** Invalidates the object in the local cache but does not distribute the operation.

  **Note:** Destruction and invalidation cause the same event notification activities whether carried out explicitly or through expiration activities.

**Region Expiration**

Expiration activities in distributed regions can be distributed or performed only in the local cache. So one cache could control region expiration for a number of caches in the distributed system.

---

**Cache Management**

This chapter covers cache management.
Client-to-Server Connection Process

It is important to understand the sequence of events that occur when the native client connects with a GemFire cache server.

1. A native client region is configured in cache.xml or programmatically with a set of server connection endpoints. Server endpoints identify each cache server by specifying the server’s name and port number.

Client threads obtain, use, and release a connection to a connection pool that maintains new connections. The number of connections that a client can establish is governed by the pool’s min-connections and max-connections settings, either using client XML configuration or programmatically through the CacheFactory::setMinConnections() and CacheFactory::setMaxConnections() APIs. The defaults for min-connections is 1 and max-connections is -1 meaning the connection count can grow to accommodate the number of active threads performing region operations.

This example shows how to use cache.xml to configure a native client region with endpoints set to two cache servers:

```
<pool name="examplePool" subscription-enabled="true" >
  <server host="java_servername1" port="java_port1" />
  <server host="java_servername2" port="java_port2" />
</pool>

<region name="NativeClientRegion" refid="CACHING_PROXY">
  <region-attributes pool-name="examplePool"/>
</region>
```

TCP connections on the native client are specified at the cache level, or by overriding endpoints for specific regions. The connections are created as the regions are created. In addition, connections can also get created for querying without having any created regions. In this case, when endpoints are defined at the cache level no regions are yet created and a query is fired.

You can configure client-server connections in two ways. Use either the region/cache endpoints or the Pool API. For more information about the pool API, see Using Connection Pools.

2. The client announces to the server which entries it wishes to have updated by programmatically registering interest in those entries. See Registering Interest for Entries for more information.

3. The client cache.xml file should have the following parameters configured so the client can update the server and the client can receive updates from the server:
   - Caching enabled in the client region, by using the CACHING_PROXY RegionShortcut setting in the region attribute refid. A listener could also be defined so event notification occurs. You can use both, but at least one of the two methods must be used by the client to receive event notifications.
   - Set subscription-enabled to true so the client receives update notifications from the server for entries to which it has registered interest.

4. A native client application calls the C++ or .NET API to connect to a cache server.

5. The client and the cache server exchange a handshake over a configured endpoint to create a connection.

6. Any create, put, invalidate, and destroy events sent to the server are propagated across the distributed cache so the client can receive the events.

   **Note:** You may be able to improve system performance by making adjustments to the cache server. For cache server tuning information, see the GemFire User’s Guide.
Controlling Cache Size

You can control cache size through region size limits, cache size limits, or a combination of the two.

GemFire controls region size by moving least recently used (LRU) entries from the region or from all cache regions.

Controlling Region Size

You can cap the size of any region with the region attribute `LruEntriesLimit`. You can specify whether to destroy the entries or overflow them to disk in the attribute `DiskPolicy`. If you overflow entries to disk, you must also specify the attribute `PersistenceManager`.

Controlling Cache Size

You can control overall cache size with the `heap-lru-limit`, which is set in `gfcpp.properties`. This property sets the maximum amount of memory used for the cache, in megabytes. If a new entry causes memory to grow past this limit, GemFire’s LRU algorithm is called to evict entries. Heap LRU causes eviction to occur on all regions in the cache, overriding region-level `LruEntriesLimit` settings when it needs to reclaim memory.

For each region, evictions are performed according to the region’s `DiskPolicy` and `PersistenceManager` settings. If you use `heap-lru-limit`, review these region attributes for all your caching regions, to be sure you are evicting the way you want to.

The related `heap-lru-delta` property, also set in `gfcpp.properties`, is the amount of memory to free up once the LRU evictions have begun. Memory is reclaimed until the amount of memory used is below `heap-lru-limit minus heap-lru-delta`. 
Managing the Lifetime of a Cached Object

All cacheable objects derive from `SharedBase`, which provides reference counting. Cacheable objects are referenced using `SharedPtr` types.

When `SharedPtr` retrieves a cached object, the object remains alive as long as that pointer or the cache itself references the object.

A native client may have many pointers that reference an object. Regardless of how many pointers to the object are deleted, the object remains alive until the last remaining pointer is deleted. At that point the object is deleted.

This is a very simple example:

```cpp
CacheableStringPtr p = CacheableString::create ("string");
region.put ("key", p);
```

In the example:
- The act of object creation allocates memory and initializes the object.
- When you assign the object to a `SharedPtr`, you relinquish control of the lifetime of that object to the reference counting mechanism for the cache.
- The put operation does not actually copy the object into the cache. Rather, it copies a `SharedPtr` into the cache’s hashmap. Consequently, the object remains alive in the cache when the original `SharedPtr` goes away.

The client can make use of an object after you have initialized the object. For example, another `SharedPtr` might issue a `get` to retrieve the object from the cache:

```cpp
CacheableStringPtr p2 = region.get ("key");
```

Because `p` (the original `SharedPtr`) and `p2` point to the same object in memory, it is possible under some circumstances for multiple `SharedPtr` types to work on the same object in data storage.

**Note:** Once you have put an object into the cache, do not delete it explicitly. Attempting to do so can produce undesirable results.

Changed Objects

If an object update is received, the cache no longer holds the same object. Rather, it holds a completely different instance of the object. The client does not see the updates until it calls a `get` to fetch the object again from the local cache, or (in a cache plug-in) calls `EntryEvent::getNewValue`.

For more about plug-ins, see Application Plug-Ins.

Object Expiration

When a cache automatically deletes an object as a result of an expiration action, the reference counting pointers protect the client from situations that might otherwise result if the cache actually freed the object's memory. Instead, the client disconnects the object from the cache by deleting the cache's `SharedPtr` reference, while leaving untouched any client threads with a `SharedPtr` to that object.

Object Lifetime Across the Distributed Cache

An object remains alive until every copy of the object is gone. In distributed regions, expiration activities can be local or distributed, depending on a region’s distribution settings. One cache could control the expiration of all copies of an object in all the caches in the distributed system. Alternatively, each cache could control the expiration of its own local copy of the object. If the configuration gives each cache local
control, and the expiration parameters are set to different lengths of time in different caches, some copies of an object may still exist after it has disappeared in other caches. See *Expiration Attributes* for more information.
Using Thread Safety in Cache Management

When you perform structural changes on your cache, such as creating or closing a Cache, Pool, or Region, synchronize your operations or do them in a single thread.

Other non-structural operations, like region gets, puts, and queries, are thread safe, and you can perform them in a multithreaded way. There are caveats to this, for example, when two threads update the same key simultaneously, there is no way to determine which thread's operation will prevail.

You may need to protect cached objects from concurrent usage and modification. The native client does not guard cached objects themselves from concurrent access.

Always catch and handle exceptions that may be thrown, for problems like trying to create a Pool with the same name more than once.
Troubleshooting

This section provides troubleshooting information for the native client.

- **Cannot Acquire Windows Performance Data**
- **Generating a Process Memory Dump Image for Fatal Errors**

### Cannot Acquire Windows Performance Data

When you attempt to run performance measurements for the native client on Windows, you may encounter the following error message in the run logs:

```
Can't get Windows performance data. RegQueryValueEx returned 5
```

This can occur because incorrect information is returned when a Win32 application calls the ANSI version of `RegQueryValueEx` Win32 API with `HKEY_PERFORMANCE_DATA`. This error is described in Microsoft KB article ID 226371 at [http://support.microsoft.com/kb/226371/en-us](http://support.microsoft.com/kb/226371/en-us).

To successfully acquire Windows performance data, you need to verify that you have the proper registry key access permissions in the system registry. In particular, make sure that `Perflib` in the following registry path is readable (`KEY_READ` access) by the GemFire process:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Perflib
```

An example of reasonable security on the performance data would be to grant administrators `KEY_ALL_ACCESS` and interactive users `KEY_READ` access. This particular configuration prevents non-administrator remote users from querying performance data.

See [http://support.microsoft.com/kb/310426](http://support.microsoft.com/kb/310426) and [http://support.microsoft.com/kb/146906](http://support.microsoft.com/kb/146906) for instructions about how to ensure that GemFire processes have access to the registry keys associated with performance.

### Generating a Process Memory Dump Image for Fatal Errors

You can generate a process memory dump image (core files in Unix systems and minidumps in Windows). The image is produced when a fatal error occurs that normally terminates the program.

When the system property `crash-dump-enabled` is set to `true`, a dump image is generated (the default is `true`). The dump file is generated in the same location as the `log-file` directory, and has the same prefix as the log file. The name is `<prefix>-<time>.core.<pid>` in Unix, and `<prefix>-<time>-<pid>.dmp` in Windows.

Unix systems generate core files automatically for such errors, but this option is useful for providing a custom location and name, as well as for systems where core dump generation is disabled. For Unix, when system core dump generation is turned on via `ulimit -c`, this property can be set to `false`.

For .NET clients, when this property is set then `AccessViolation` exceptions are trapped and a crash dump is created to assist with further analysis. Applications receive a `FatalInternalException` for this case, with the `InnerException` set to the originating `AccessViolationException`.

This requires the availability of `dbghelp.dll` on Windows, either in the same directory as `gfcppcache.dll` or in the system `PATH`. The file is installed by default, though for Windows 2000 a newer version may be required for minidumps. For Unix systems, the `gcore` command should be available (gdb > 5.2 on Linux; available by default in Solaris).
Part VIII

Pivotal GemFire Native Client C++ API

Pivotal GemFire Native Client C++ API describes the primary classes, and usage conventions for the native client C++ API. It demonstrates how to use the API to create caches and perform data serialization.

The native client C++ API allows C++ and .NET developers to programmatically create, populate, and manage a Pivotal GemFire distributed system. The C++ library is thread-safe, except where specified otherwise in the online API documentation.

The online C++ API documentation is included in the docs/cppdocs directory of the GemFire native client product installation. It provides extensive implementation details for the C++ structures and functions.

Several example API programs are included in the SampleCode directory. See QuickStart Examples and Guide.

About the Native Client C++ API

The native client C++ API allows C++ and .NET developers to programmatically create, populate, and manage a Pivotal GemFire distributed system. The C++ library is thread-safe, except where specified otherwise in the online API documentation.

This chapter gives a general overview of the classes in the gemfire, gemfire_statistics, and gemfire_admin namespaces. For complete and current information on the classes listed here, see the online C++ API documentation.

• Cache Classes
• Region Classes
• Region Attribute Classes
• Application Plug-In Classes
• Event Handling Classes
• Statistics API

Cache Classes

The native client C++ API has the following cache classes:

• CacheFactory. Use this class to create and configure a Cache instance. If cache.xml is specified, the cache is created based on the declarations loaded from that file.
• Cache. Entry point to the client caching API. The cache is created by calling the create function of the factory class, CacheFactory. Regions are configured and obtained using the Cache::createRegionFactory() API.

Region Classes

The native client C++ API has the following region classes:

• Region. Provides functions for managing regions and cached data. Use these functions to perform the following actions:
- Retrieve information about the region, such as its parent region and region attribute objects.
- Invalidate or destroy the region.
- Create, update, invalidate and destroy region entries.
- Retrieve region entry keys, entry values, and RegionEntry objects, either individually or as entire sets.
- Retrieve the statistics object associated with the region.
- Set and get user-defined attributes.
- **RegionEntry.** Contains the key and value for the entry, and provides all non-distributed entry operations. This object's operations are not distributed and do not affect statistics.

### Region Attribute Classes
The native client C++ API has the following region attribute classes:

- **RegionAttributes.** Holds all attribute values for a region and provides functions for retrieving all attribute settings. This class can be modified by the AttributesMutator class after region creation.
- **AttributesMutator.** Allows modification of an existing region's attributes for application plug-ins and expiration actions. Each region has an AttributesMutator instance.

### Application Plug-In Classes
The native client C++ API has the following application plug-in classes:

- **CacheLoader.** Loads data into a region on a cache miss.
- **CacheWriter.** Synchronously handles region and entry events before the events occur. Entry events are create, update, invalidate, and destroy. Region events are invalidate and destroy. This class has the ability to abort events.
- **CacheListener.** Handles region and entry events after they occur. Entry events are create, update, invalidate, and destroy. Region events are invalidate and destroy.

### Event Handling Classes
The native client C++ API has the following event handling classes:

- **RegionEvent.** Provides information about the event, such as what region the event originated in, whether the event originated in a cache remote to the event handler, and whether the event resulted from a distributed operation.
- **EntryEvent.** Provides all available information for the RegionEvent, and provides entry-specific information such as the old and new entry values and whether the event resulted from a load operation.

### Statistics API
The StatisticsType API represents a blueprint for the same type of Statistics. The StatisticsType API is a collection of StatisticDescriptor. Internally, each StatisticDescriptor describes data of each individual statistic. StatisticsFactory provides functionality for creating StatisticDescriptor, StatisticsType, and Statistics.

- **CacheStatistics.** This class defines common statistics functions. Region and RegionEntry both have functions that return a CacheStatistics object for accessing and resetting their statistics counts.
- **StatisticDescriptor.** An instance of this class describes a statistic whose value is updated by an application and may be archived by the native client. Each statistic has a type of either int, long, or double, and either a gauge or a counter. The value of a gauge can increase and decrease, and the value of a counter strictly increases. Create an instance of StatisticDescriptor by calling one of these StatisticsFactory functions: createDoubleCounter, createDoubleGauge, createIntCounter, createIntGauge ], createLongCounter, createLongGauge.
Creating a Cache

The code snippets in this section show cache creation.

When you create your cache, the system automatically connects your process to the server tier. For systems with security enabled, the credentials for a connecting client are authenticated when it creates the cache. See Security for more information about authenticated connections.

```
Creating the System Connection and the Cache
In this example, the application creates the cache by calling the CacheFactory::create function, specifying the servers to connect to:

CacheFactoryPtr cacheFactory = CacheFactory::createCacheFactory();
CachePtr cachePtr = cacheFactory
  ->addServer("localhost", 40404)
  ->addServer("localhost", 40405)
  ->setSubscriptionEnabled(true)
  ->create();
```

Creating a Proxy Client-Side Region

This section provides code examples for creating and customizing proxy client-side regions.

**Note:** Creating a region through the native client API only creates a proxy client-side region. A corresponding region with the same name and path should also exist on the servers that have been configured for client connections and upon which the client will perform its operations.

To create a region, you create a RegionFactory using the RegionShortcut that most closely fits your region configuration. From that, create your region, customizing the settings as region attributes as needed.

```
Creating a CACHING_PROXY Region
This example creates a region using a CACHING_PROXY RegionShortcut with no further modifications:

RegionFactoryPtr regionFactory = cachePtr->createRegionFactory(CACHING_PROXY);
regionPtr = regionFactory ->create("exampleRegion");
```
Creating a CACHING_PROXY Region with LRU

This example creates a region based on the CACHING_PROXY RegionShortcut with two additional region attributes settings. For information on the settings, see Region Attributes Descriptions.

RegionFactoryPtr regionFactory = cachePtr->createRegionFactory(CACHING_PROXY);
regionPtr = regionFactory->setLruEntriesLimit( 20000 )
->setInitialCapacity( 20000 )
->create("exampleRegion");

Adding an Entry to the Cache

You can populate a native client region with cache entries using the Region::put or the Region::create API functions. Code examples demonstrate these actions.

The put function places a new value into a region entry with the specified key, while the create function creates a new entry in the region. Both functions provide a user-defined parameter object to any cache writer invoked in the process, and new values for both functions are propagated to a connected cache server.

Adding Entries Using create

When the put function adds an entry, the previous value is overwritten if there is already an entry associated with the specified key in the region. In the next example, the program uses the API to put 100 entries into the cache by iteratively creating keys and values, both of which are integers.

for ( int32_t i=0; i < 100; i++ ) {
    regionPtr->put( i, CacheableInt32::create(i) );
}

Bulk Put Operations Using putAll

You can batch up multiple key/value pairs into a hashmap and put them into the cache with a single operation using the Region::putAll API function (Region.PutAll for .NET). Each entry is processed for interest registration on the server, so each entry requires its own unique event ID. Updates and creates can be mixed in a putAll operation, so those events need to be addressed on the cache server for appropriate cache listener invocation on distributed system members. Map entries retain their original order when they are processed at the server.

The following table lists the client and cache server statistics for putAll.

Table 14: PutAll Statistics for Cache Server and Client

<table>
<thead>
<tr>
<th>Statistic Type</th>
<th>Chart Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CachePerfStats</td>
<td>Putalls</td>
<td>Total number of times a map is added or replaced in the cache as a result of a local operation. Also reports the number of putAll operations.</td>
</tr>
</tbody>
</table>
### Accessing an Entry

The putAll API also supports providing a callback argument to any cache loaders or cache writers that are invoked in the operation.

#### Bulk Get Operations Using getAll

You can use the `Region::getAll` API (Region.GetAll for .NET) to get values for an array of keys from the local cache or server. If the value for a key is not present locally, then it is requested from the server.

**Note:** The value returned is not copied, so multi-threaded applications should not modify the value directly, but should instead use the update methods.

The `getAll` API also supports providing a callback argument to any cache loaders or cache writers that are invoked in the operation. See the Region online API documentation for more information about using `getAll`.

<table>
<thead>
<tr>
<th>Statistic Type</th>
<th>Chart Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheperfStats</td>
<td>putallTime</td>
<td>Total time to replace a map in the cache as a result of a local operation.</td>
</tr>
<tr>
<td>CacheServerStats</td>
<td>putAllRequests</td>
<td>Number of putAll requests.</td>
</tr>
<tr>
<td>CacheServerStats</td>
<td>putAllResponses</td>
<td>Number of putAll responses written to the cache client.</td>
</tr>
<tr>
<td>CacheServerStats</td>
<td>processPutAllTime</td>
<td>Total time to process a cache client putAll request, including the time to put all objects into the cache.</td>
</tr>
<tr>
<td>CacheServerStats</td>
<td>readPutAllRequestTime</td>
<td>Total time to read putAll requests.</td>
</tr>
<tr>
<td>CacheServerStats</td>
<td>writePutAllResponseTime</td>
<td>Total time to write putAll responses.</td>
</tr>
<tr>
<td>CacheClientStats</td>
<td>putAll</td>
<td>Number of putAll requests sent to the cache server.</td>
</tr>
<tr>
<td>CacheClientStats</td>
<td>sendPutAllTime</td>
<td>Total time for sendPutAll.</td>
</tr>
</tbody>
</table>

The `putAll` API also supports providing a callback argument to any cache loaders or cache writers that are invoked in the operation.
Removing an Entry

The standard Region::remove API removes the entry with specified key and provides a user-defined parameter object to any CacheWriter or CacheListener invoked in the process.

The remove call not only removes the value, but also the key and entry from this region. The remove operation is propagated to the GemFire cache server that the native client is connected to. If the destroy operation fails due to an exception on the server (for example, a CacheServerException or security exception), then the local entry is still removed.

The remove operation updates CacheStatistics::getLastAccessedTime and CacheStatistics::getLastModifiedTime for this region and the entry.

The remove API returns true if the entry (key, value) has been removed or false if the entry (key, value) has not been removed.

Bulk Remove Operations Using removeAll

You can use the Region::removeAll API to remove all entries for a collection of specified keys from the region. The effect of this call is equivalent to that of calling destroy on this region once for each key in the specified collection. If an entry does not exist, then that key is skipped. Note that an EntryNotFoundException is not thrown.

The removeAll API also supports providing a callback argument to any cache loaders or cache writers that are invoked in the operation. See the Region online API documentation for more information about using removeAll.

Serializing Data

All data that Pivotal GemFire moves out of the local cache must be serializable.
Region Data Requiring Serialization

Certain region types (including client regions) require serialization.

Region data in the following types of regions must be serializable:

- Partitioned regions (except functions that add data locally to a partitioned region use the deserialized form).
- Distributed regions.
- Regions that are persisted or overflowed to disk.
- Server or client regions in a client/server installation.
- Regions distributed between gateways in a multi-site installation.
- Regions that receive events from remote caches.
- Regions that provide function arguments and results.

To minimize the cost of serialization and deserialization, GemFire avoids changing the data format whenever possible. This means your data may be stored in the cache in serialized or deserialized form, depending on how you use it. For example, if a server acts only as a storage location for data distribution between clients, it makes sense to leave the data in serialized form, ready to be transmitted to clients that request it. Partitioned region data is always stored in serialized form with one exception—functions that add data to a partitioned region locally use the deserialized form.
Data Serialization Options

The native client C++ API gives you two serialization options: the GemFire::Serializable interface and GemFire PDX serialization.

GemFire Portable Data eXchange (PDX) serialization is the recommended option. GemFire PDX serialization provides portability for PDX serializable objects so that clients can share data with Java servers and other non-C++ clients. PDX is a cross-language data format that can reduce the cost of distributing and serializing your objects. PDX stores data in named fields that you can access individually in order to avoid the cost of deserializing the entire data object. PDX also allows you to mix versions of objects where you have added or removed fields.

When using PDX serialization, you can either use the PdxSerializer (for all your domain objects) or PdxSerializable (for a specific domain object).

PdxSerializer is used when a user has registered his or her domain class for serialization in the cache using the registerPdxSerializer API.

PdxSerializable is used when the domain class that a user wants to serialize/deserialize is inherited from PdxSerializable interface, and the user has registered domain class using registerPdxType(domainClass) API.

The non-PDX serialization option is to use the GemFire::Serializable interface. The GemFire::Serializable interface can be a good option performance-wise if the size of your objects is small. The GemFire::Serializable is used whenever a user domain class is not inherited by PdxSerializable but the user has registered his or her class with the registerType API. (See Serializing Data with the GemFire Serializable Interface for more information.)

Table 15: Serialization Options: Comparison of Features

<table>
<thead>
<tr>
<th>Capability</th>
<th>GemFire::Serializable</th>
<th>GemFire::PdxSerializable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handles multiple versions of domain objects*</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Provides single field access on servers of serialized data, without full deserialization. Supported also for OQL queries.</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Automatically ported to other languages by GemFire - no need to program Java-side implementation</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Works with GemFire delta propagation</td>
<td>X</td>
<td>X (See Using PDX Serialization with Delta Propagation for requirements.)</td>
</tr>
</tbody>
</table>

*You can mix domain object versions where the differences between versions are the addition and removal of object fields.

For detailed information on the interfaces, see the online API documentation.
Serializing Data with GemFire PDX Serialization

PDX is a cross-language data format that can reduce the cost of distributing and serializing your objects. PDX stores data in named fields that you can access individually to avoid the cost of deserializing the entire data object. When you use GemFire PDX serialization with the native client C++ API, you can register a PdxSerializer for the entire cache, implement PDX serialization for each domain object or use automatic PDX serialization by running the pdxautoserializer tool.

You can also set the object preference of the cache to the PdxInstance type, which allows you to access fields of a PDX object without deserializing the entire object.

When using the native client C++ API, you can opt to use PDX autoserialization. The GemFire native client provides a command line tool called pdxautoserializer that will automatically generate C++ code that will PDX serialize the class you want to serialize.

Serialize Your Domain Objects with PdxSerializer and PdxWrapper

For domain objects that you cannot or do not want to modify, use the PdxSerializer and the PdxWrapper classes to serialize and deserialize the object's fields.

You register a PdxSerializer implementation for the entire cache, programming it for all of the domain objects that you handle in this way. This way you do not have to implement the PdxSerializable interface for each domain class.

The PdxSerializer allows domain classes to be serialized and deserialized as PDXs without modification of the domain class. It only requires that the domain class have a constructor accessible to the PdxSerializer to create an instance. The domain class will be held in a wrapper class PdxWrapper.

PdxSerializer has the following methods:

- The toData method returns true if the PdxSerializer was able to serialize userObject, false if not.
- If the PdxSerializer was able to deserialize the object, the fromData method returns a void pointer to the user object to be wrapped in a PdxWrapper.

When you later reference the user object, use the PdxWrapper class. PdxWrapper holds a shared reference to the object in the local cache and is used during serialization and deserialization. PdxWrapper acts as a container for the user domain object and needs to wrap every instance of the object that uses a registered PdxSerializer. The object instance will not be modified. In addition, when using PdxWrapper, you will need to provide a function pointer to a "de-allocator" which will delete the user object when the reference is no longer held.

The following code example defines a user object and a PdxSerializer. It then registers the new PdxSerializer and then uses PdxWrapper to put the object in a region and retrieve the object from a region.

```cpp
class UserClass
{
public:
  int m_int;
  string m_string;

  UserClass(int intVal, string stringVal)
  {
    m_int = intVal;
    m_string = stringVal;
  }

  static void deallocate(void * object, char * className)
```
{  
    if (strcmp(className, "com.example.UserClass") == 0) {
        UserClass * userObject = reinterpret_cast<UserClass*>(object);
        delete userObject;
    }
}
};

class UserPdxSerializer : public PdxSerializer {
public:
    void* fromData(char * className, PdxReaderPtr pdxReader) {
        if (strcmp(className, "com.example.UserClass") != 0) {
            return NULL;
        }
        int intVal = pdxReader->readInt("m_int");
        string stringVal = pdxReader->readString("m_string");
        UserClass * userObject = new UserClass(intVal, stringVal);
        return (void*) userObject;
    }

    bool toData(void * object, char * className, PdxWriterPtr pdxWriter) {
        if (strcmp(className, "com.example.UserClass") != 0) {
            return false;
        }
        UserClass * userObject = reinterpret_cast<UserClass*>(object);
        pdxWriter->writeInt("m_int", userObject->m_int);
        pdxWriter->writeString("m_string", userObject->m_string);
        return true;
    }

    UserDeallocator getDeallocator(char * className) {
        if (strcmp(className, "com.example.UserClass") == 0) {
            return UserClass::deallocate;
        } else {
            return NULL;
        }
    }
};

// Register a user PDX serializer
Serializable::registerPdxSerializer(new UserPdxSerializer);

// Put a user object into a region.
UserClass * userObject = new UserClass(123, "someValue");
PdxWrapperPtr pdxWrapper = new PdxWrapper(userObject, "com.example.UserClass", UserClass::deallocate);
region->put("key", pdxWrapper);

// Get a user object from a region.
pdxWrapper = dynCast<PdxWrapperPtr>(region->get("key"));
UserClass * userObject = reinterpret_cast<UserClass*>(pdxWrapper->getObject());
Serialize Using the GemFire PdxSerializable Class

Domain classes need to inherit the PdxSerializable abstract class to serialize and de-serialize the object. When you write objects using PDX serialization, they are distributed to the server tier in PDX serialized form.

When you run queries against the objects on the servers, only the fields you specify are deserialized. A domain class should serialize and de-serialize all its member fields in the same order in its toData and fromData method.

Use this procedure to program your domain object for PDX serialization using the PdxSerializable abstract class.

1. In your domain class, implement PdxSerializable. Example:

   ```
   class PdxObject: public PdxSerializable
   
   private:
     uint32_t m_id;
     char* m_str;
   
   public:
     PdxObject(){);
     PdxObject(uint32_t id, char* str);
     virtual ~PdxObject();
     
     uint32_t getID() { return m_id; }
     char* getStr(){ return m_str; }
   
   virtual void toData(PdxWriterPtr pw) const;
   virtual void fromData(PdxReaderPtr pr);
   CacheableStringPtr toString() const;
   virtual char* getClassName() const;
   static Cacheable* createDeserializable() {
     return new PdxObject();
   }
   ```

2. Program the toData function to serialize your object as required by your application.

   If you also use PDX serialization in Java or .NET for the object, serialize the object in the same way for each language. Serialize the same fields in the same order and mark the same identity fields.

3. Program the fromData method to read your data fields from the serialized form into the object's fields.

   In your fromData implementation, use the same name as you did in toData and call the read operations in the same order as you called the write operations in your toData implementation.

4. Optionally, program your domain object's hashCode and equals methods.

   Use the markIdentityField method to indicate that the given field name should be included in hashCode and equals checks of this object on a server.

   The fields that are marked as identity fields are used to generate the hashCode and equals methods of PdxInstance. Because of this, the identity fields should themselves either be primitives, or implement hashCode and equals.

   If no fields are set as identity fields, then all fields will be used in hashCode and equals checks. The identity fields should make marked after they are written using a write* method.
Performing put, get, and localDestroy Operations with a PDX Domain Object

This topic demonstrates how you can perform operations on a PDX domain object after you have implemented PDX serializable in your domain class.

For example, you can perform operations like put, get, and localDestroy with the domain class you defined for PDX serialization in the PdxSerializable Example.

To perform operations, you could write the following application code:

1. Register the PDX domain class.

   ```
   Serializable::registerPdxType(PdxObject::createDeserializable);
   ```

2. Create the PDX domain object PdxObject.

   ```
   CacheablePtr pdxobj(new PdxObject(100, "Value-1"));
   CacheableKeyPtr keyport = CacheableKey::create("ABC");
   ```

3. Here's an example of a put operation.

   ```
   rptr->put(keyport, pdxobj);
   ```

4. Here's an example of locally destroying the entry.

   ```
   rptr->localDestroy(keyport);
   ```

5. Here's an example of a get operation.

   ```
   PdxObject *obj2 = dynamic_cast<PdxObject *>(rptr->get(keyport)).ptr();
   ```
Using Automatic PDX Serialization

You can allow your native client C++ applications to automatically PDX serialize and deserialize domain objects without having to add any extra code by using the `pdxautoserializer` command line tool provided with the native client.

When using the native client C++ API, you can automatically serialize and deserialize domain objects without making any code changes to those objects or having to implement a `PdxSerializer` or `PdxSerializable` interface and their related `fromData` and `toData` methods. The GemFire native client includes a command-line utility, `pdxautoserializer`, that allows you to generate C++ code that will serialize your domain objects in the PDX format for you.

Programming Your Application to Use PdxInstances

A `PdxInstance` is a lightweight wrapper around the raw bytes of the PDX serialized objects kept in the cache. It provides applications with run-time access to files of a PDX serialized object. GemFire provides the implementation of the `PdxInstance` class.

You can configure your cache to return a `PdxInstance` when a PDX serialized object is deserialized instead of deserializing the object to a domain class. Preventing deserialization saves both time and memory and does not require you deserialize the object to the domain class.

This configuration can be done in cache.xml by setting the attribute `read-serialized` to `true` on the `<pdx>` element. Or it can be done programmatically using the `CacheFactory::setPdxReadSerialized(bool)` method.

After this preference is configured, any time a PDX object is deserialized, it is deserialized into a `PdxInstance`.

The following is a code sample of using the `setField` API of `PdxInstance` to modify fields:

```cpp
RegionPtr rptr = getHelper()->getRegion( regionNames[0] );
CacheableKeyPtr keyport = CacheableKey::create("pdxput");
CacheableKeyPtr keyport1 = CacheableKey::create("pdxput2");
PdxInstancePtr pIPtr = dynCast<PdxInstancePtr>(rptr->get(keyport));
LOG( "modifyPdxInstance get complete." );
WritablePdxInstancePtr wpiPtr( pIPtr->createWriter());
ASSERT(pIPtr != NULLPTR, "pIPtr != NULLPTR expected");
int val = 0;
int newVal = 0;
ASSERT(pIPtr->hasField("m_int32") == true, "m_id1 = true expected");
pIPtr->getField("m_int32", val);
pIPtr->setField("m_int32", val + 1);
rptr->put(keyport, wpiPtr);
PdxInstancePtr newPiPtr = dynCast<PdxInstancePtr>(rptr->get(keyport));
ASSERT(newPiPtr->hasField("m_int32") == true, "m_int32 = true expected");
newPiPtr->getField("m_int32", newVal);
ASSERT(val + 1 == newVal, "val + 1 == newVal expected");
ASSERT((pIPtr.ptr() == *newPiPtr.ptr()) == false, 
"PdxInstance should not be equal");
```

In addition to field access, `PdxInstance` also supports field modification using the `setField(fieldName)` method. The `setField` method has copy-on-write semantics. So for the modifications to be stored in the cache, the `PdxInstance` must be put into a region after `setField` has been called one or more times.
Configuring PDX to Ignore Unread Fields During Deserialization

Use the `setPdxIgnoreUnreadFields` API to control whether PDX ignores fields that were unread during deserialization.

The default is to preserve unread fields by including their data during serialization. However, if you configure the cache to ignore unread fields then their data will be lost during serialization.

You should only set this attribute to `true` if you know this member will only be reading cache data. In this use case you do not need to pay the cost of preserving unread fields since you will never reserialize the PDX data.

For example:

```cpp
CacheFactoryPtr cfPtr = CacheFactory::createCacheFactory(PropertiesObj);
cfPtr->setPdxReadSerialized(true);
cfPtr->setPdxIgnoreUnreadFields(false);
cachePtr = cfPtr->create();
```

Using PdxInstanceFactory to Create PdxInstances

You can use the `PdxInstanceFactory` API to create a `PdxInstance` from raw data when the domain class is not available on the server.

Creating a `PdxInstance` can be particularly useful when you need an instance of a domain class for plug-in code such as a function or a loader. If you have raw data for the domain object (the class name and each field's type and data), then you can explicitly create a `PdxInstance`. The `PdxInstanceFactory` API is very similar to the `PdxWriter` API except that after writing each field, you need to call the create method which returns the created `PdxInstance`.

PdxInstance Example

The following is a code example of creating a `PdxInstance`.

```cpp
class Person
{
private:
    char* m_name;
    int m_id;
    int m_age;

public:
    Person();
    Person(char* name, int id, int age)
    {
        m_name = name;
        m_id = id;
        m_age = age;
    }

    char* getName() const
    {
        return m_name;
    }
    int getId()
    {
        return m_id;
    }
    int getAge()
    {
        return m_age;
    }
};
```
int main(int argc, char ** argv)
{
    try
    {
        // Create a GemFire Cache.
        CacheFactoryPtr cacheFactory = CacheFactory::createCacheFactory();

        CachePtr cachePtr = cacheFactory->set("cache-xml-file", "XMLs/clientPdxInstance.xml")->create();
        LOGINFO("Created the GemFire Cache");

        // Get the example Region from the Cache which is declared in the Cache XML file.
        RegionPtr regionPtr = cachePtr->getRegion("Person");
        LOGINFO("Obtained the Region from the Cache.");

        Person* p = new Person("Jack", 7, 21);
        //PdxInstanceFactory for Person class
        PdxInstanceFactoryPtr pif = cachePtr->createPdxInstanceFactory("Person");
        LOGINFO("Created PdxInstanceFactory for Person class");

        pif->writeString("m_name", p->getName());
        pif->writeInt("m_id", p->getID());
        pif->markIdentityField("m_id");
        pif->writeInt("m_age", p->getAge());

        PdxInstancePtr pdxInstance = pif->create();
        LOGINFO("Created PdxInstance Object");

        regionPtr->put("Key1", pdxInstance);
        LOGINFO("Populated PdxInstance Object");

        PdxInstancePtr retPdxInstance = regionPtr->get("Key1");
        LOGINFO("Got PdxInstance Object");

        int id = 0;
        retPdxInstance->getField("m_id", id);

        int age = 0;
        retPdxInstance->getField("m_age", age);

        char* name = NULL;
        retPdxInstance->getField("m_name", &name);

        if(id == p->getID() && age == p->getAge() && strcmp(name, p->getName()) == 0
            && retPdxInstance->isIdentityField("m_id") == true)
            LOGINFO("PdxInstance returns all fields value expected");
        else
            LOGINFO("PdxInstance doesn't return all fields value expected");

        delete p;

        // Close the GemFire Cache.
        cachePtr->close();
        LOGINFO("Closed the GemFire Cache");
    }
    // An exception should not occur
    catch(const Exception & gemfireExcp)
Using C++ Enum Type with PDX Serialization

Because there is no "object" base type in C++, enums cannot be directly passed as a parameter to the writeObject and readObject API.

To use the C++ enum type with PDX serialization, you will have to wrap the enum in the CacheableEnum class type by specifying class name, enum name and ordinal.

```cpp
class TESTOBJECT_EXPORT PdxEnumTestClass :public PdxSerializable
{
private:
  int m_id;
  CacheableEnumPtr m_enumid;

public:
  int getID(){
    return m_id;
  }

  CacheableEnumPtr getEnumID() {
    return m_enumid;
  }

  PdxEnumTestClass(int id)
  {
    m_id = id;
    switch (m_id)
    {
      case 0:
        m_enumid = CacheableEnum::create("enumQuerytest", "id1", id1);
        break;
      case 1:
        m_enumid = CacheableEnum::create("enumQuerytest", "id2", id2);
        break;
      case 2:
        m_enumid = CacheableEnum::create("enumQuerytest", "id3", id3);
        break;
      default:
        m_enumid = CacheableEnum::create("enumQuerytest", "id1", id1);
        break;
    }
  }

  PdxEnumTestClass() {}

  void toData(PdxWriterPtr pw) {
    pw->writeInt("m_id", m_id);
    pw->writeObject("m_enumid", m_enumid);
  }

  void fromData(PdxReaderPtr pr) {
    m_id = pr->readInt("m_id");
    m_enumid = pr->readObject("m_enumid");
  }

  CacheableStringPtr toString() const {
    return CacheableString::create("PdxEnumTestClass");
  }

  char* GetClassName() const {
    return "com.example.PdxEnumTestClass";
  }
};
```
How Put and Queries Work on Enum

The following code sample demonstrates how put and query operations work when using the C++ enum Type with PDX serialization:

```cpp
//Creating objects of type PdxEnumTestClass
PdxEnumTestClassPtr pdxobj1(new PdxEnumTestClass(0));
PdxEnumTestClassPtr pdxobj2(new PdxEnumTestClass(1));
PdxEnumTestClassPtr pdxobj3(new PdxEnumTestClass(2));
RegionPtr rptr = getHelper()->getRegion( "DistRegionAck" );
//PUT Operations
rptr->put( CacheableInt32::create(0), pdxobj1 );
LOG("pdxPut 1 completed ");
rptr->put( CacheableInt32::create(1), pdxobj2 );
LOG("pdxPut 2 completed ");
rptr->put( CacheableInt32::create(2), pdxobj3 );
LOG("pdxPut 3 completed ");
//Query
try {
    Serializable::registerPdxType(PdxEnumTestClass::createDeserializable);
    LOG("PdxEnumTestClass Registered Successfully....");
} catch (gemfire::IllegalStateException&/* ex*/) {
    LOG("PdxEnumTestClass IllegalStateException");
}
RegionPtr rptr = getHelper()->getRegion( "DistRegionAck" );
SelectResultsPtr results = rptr->query("m_enumid.name = 'id2' ");
ASSERT(results->size()== 1 , "query result should have one item");
ResultSetPtr rsptr = dynCast<ResultSetPtr>(results);
SelectResultsIterator iter = rsptr->getIterator();
while (iter.moveNext())
{
    PdxEnumTestClassPtr re = dynCast<PdxEnumTestClassPtr>(iter.current());
    ASSERT(re->getID()== 1 , "query should have return id 1");
}
```

Using PDX Serialization with Delta Propagation

To use delta propagation with PDX serialization, you must implement the Delta interface methods.

Users can include delta propagation support with PDX serialization by implementing the Delta interface methods. However, using delta propagation with PDX will require the users to implement Java side classes and the objects will remain in deserialized form at all times on the server and you will lose one of the main benefits of PDX.

The following code snippet is a sample implementation of the Delta interface methods for using with PDX serialization.

```cpp
class PdxWithDelta : public PdxSerializable, public Delta
{
public:
    bool hasDelta();
```
void toDelta(DataOutput& output);
void fromDelta(DataInput& input);
DeltaPtr clone();

// other PdxSerializable methods here...
Serializing Data with the GemFire Serializable Interface

The native client C++ API provides a `Serializable` interface that you can use for fast and compact data serialization. This section discusses the GemFire serializable interface, and presents implementation examples.

- **How Serialization Works**
- **Implementing the Serializable Interface**
- **Example 1: The Simple Class BankAccount**
- **Example 2: Implementing a Serializable Class**
- **Registering the Type**
- **Custom Key Types**
- **Example 3: Extending a Serializable Class To Be a CacheableKey**
- **Serialization in Native Client Mode with a Java Server**

### How Serialization Works

When your application puts an object into the cache for subsequent distribution, GemFire serializes the data by taking these steps:

1. Calls the appropriate `classId` function.
2. Writes the full `typeId` using the `classId` for the instance.
3. Invokes the instance's `toData` function.

When your application subsequently receives a byte array, GemFire takes the following steps:

1. Decodes the `typeId`, extracts the `classId` from the `typeId`, then creates an object of the designated type using the registered factory functions.
2. Invokes the `fromData` function with input from the data stream.
3. Decodes the data, then populates the data fields.

### Implementing the Serializable Interface

To store your own data types in the cache, you need to derive a new subclass from the `Serializable` interface. In practical terms, this means that you need to implement a small set of helper functions:

1. **Write a `toData` function that serializes your data.**

   ```cpp
   void toData (DataOutput& output)
   ```

   The `toData` function is responsible for copying all of the object’s data fields to the object stream.

   The `DataOutput` class represents the output stream and provides methods for writing the primitives in a network byte order. For more about this, see the online API documentation for `DataOutput`.

2. **Write a `fromData` function that consumes a data input stream and repopulates the object’s data fields.**

   ```cpp
   void fromData (DataInput& input)
   ```

   The `fromData` function must read the elements of the input stream in the same order that they were written by `toData`. For more about this, see the online API documentation for `DataInput`. 

---

123
Example 1: The Simple Class BankAccount

This example demonstrates a simple BankAccount class that encapsulates two ints: ownerId and accountId:

```cpp
class BankAccount
{
    private:
        int m_ownerId;
        int m_accountId;

    public:
        BankAccount( int owner, int account )
        : m_ownerId( owner ),
        m_accountId( account )
        {}

        int getOwner( )
        {
            return m_ownerId;
        }

        int getAccount( )
        {
            return m_accountId;
        }
};
```

To make BankAccount serializable, you would need to derive the class from Serializable and implement the following:

- toData—a function to serialize the data.
- fromData—a function to deserialize the data.
- classId—a function to provide a unique integer for the class.
- TypeFactoryMethod—a pointer to a function that returns a Serializable* to an uninitialized instance of the type.

Example 2: Implementing a Serializable Class

This example shows a code sample that demonstrates how to implement a serializable class.

```cpp
class BankAccount : public Serializable
{
    private:
        int m_ownerId;
        int m_accountId;

    public:
        BankAccount( int owner, int account )
        : m_ownerId( owner ),
        m_accountId( account )
        {}

        int getOwner( )
        {
            return m_ownerId;
        }

        int getAccount( )
        {
            return m_accountId;
        }
};
```
// Add the following for the Serializable interface
// Our TypeFactoryMethod
static Serializable* createInstance( )
{
    return new BankAccount( 0, 0 );
}
int32_t classId( )
{
    return 10; // must be unique per class.
}
virtual uint32_t objectSize() const
{
    return 10;
}
void toData( DataOutput& output )
{
    output.writeInt( m_ownerId );
    output.writeInt( m_accountId );
}
Serializable* fromData( DataInput& input )
{
    input.readInt( &m_ownerId );
    input.readInt( &m_accountId );
    return this;
};

Registering the Type
To be able to use the BankAccount type, you must register it with the type system so that when an incoming stream contains a BankAccount, it can be manufactured from the associated TypeFactoryMethod.

Serializable::registerType( BankAccount::createInstance );

Typically, you would register the type before calling the function DistributedSystem::connect.

Note: Type IDs must be unique to only one class.

Custom Key Types
If your application uses key types that are too complex to easily force into CacheableString, you can likely improve performance by deriving a new class from CacheableKey. If you have hybrid data types you can implement your own derivation of CacheableKey that encapsulates the data type.

See Serialization in Native Client Mode with a Java Server for information about implementing key types for a native client that is used with a Java cache server.

To extend a Serializable class to be a CacheableKey, you need to modify the class definition as follows:

- Change the class so that it derives from CacheableKey rather than Serializable.
- Implement operator== and hashcode functions.

The next example demonstrates how to extend a serializable class to be a cacheable key.

Example 3: Extending a Serializable Class To Be a CacheableKey
This example shows how to extend a serializable class to be a cacheable key.

class BankAccount
 : public CacheableKey
{
    private:
        int m_ownerId;
```cpp
int m_accountId;
public:
    BankAccount( int owner, int account )
    : m_ownerId( owner ),
    m_accountId( account )
    {
    }
    int getOwner( )
    {
        return m_ownerId;
    }
    int getAccount( )
    {
        return m_accountId;
    }
    // Our TypeFactoryMethod
    static Serializable* createInstance( )
    {
        return new BankAccount( 0, 0 );
    }
    int32_t typeId( )
    {
        return 1000; // must be unique per class.
    }
    void toData( DataOutput& output )
    {
        output.writeInt( m_ownerId );
        output.writeInt( m_accountId );
    }
    Serializable* fromData( DataInput& input )
    {
        input.readInt( &m_ownerId );
        input.readInt( &m_accountId );
        return this;
    }
    // Add the following for the CacheableKey interface
    bool operator == ( const CacheableKey& other ) const
    {
        const BankAccount& otherBA =
        static_cast<const BankAccount&>( other );
        return (m_ownerId == otherBA.m_ownerId) &&
        (m_accountId == otherBA.m_accountId);
    }
    uint32_t hashcode( ) const
    {
        return m_ownerId;
    }
    virtual int32_t classId( )const
    {
        return 10; // must be unique per class.
    }
    virtual uint32_t objectSize() const
    {
        return 10;
    }
};
```

**Serialization in Native Client Mode with a Java Server**

Primitive object types supported in all languages (CacheableInt32, CacheableString, CacheableBytes) function without requiring custom definitions with the Java cache server. For the keys, the Java cache server has to deserialize them and locate the hashcode to be able to insert the internal maps. Because of this, key types for C++ and .NET native clients used with a Java server are required to be registered on the Java server, but the value types do not need to be registered. This needs to be done even if there are no
Java clients. The Java serializable type should have the same `classId` as the .NET class, and it should serialize and deserialize the type in the same manner as the .NET implementation.

See *Data Serialization* for more information about .NET data serialization.
Serializing Object Graphs

If you have a graph of objects where each node can be serializable, the parent node can call `DataOutput::writeObject` to delegate the serialization responsibility to its child nodes. Similarly, your application can call `DataInput::readObject` to deserialize the object graph.

For more information, see the online API documentation for `DataOutput` and `DataInput`. 
Serializing and Accessing Data as a Blob

If you have data that is best handled as a blob, such as structs that do not contain pointers, use the serializable type `CacheableBytes`. `CacheableBytes` is a blob class that implements the serialization for you.

`CacheableBytes` also provides direct access to the blob data. Because it is not derived from the `CacheableKey` interface, `CacheableBytes` enables you to modify data in place and then put it into the region again to distribute the change.

For more information, see the online API documentation for `CacheableBytes`.

Implementing User-Defined Objects in Java Clients

You can use one of two methods to implement a user-defined object in a Java client that works with C++ clients: `Instantiator.register` and `DataSerializable`.

Instantiator.register

With the `Instantiator.register` method, a client sends a `RegistrationMessage` to every Java VM in its distributed system. The message announces the mapping between a user-defined classId and class name. The other JVMs can deserialize the byte array with the correct class.

If two clients are in different distributed systems, a `RegistrationMessage` cannot be sent between them. For example: a `put` made by a client in one distributed system will hang when a client in another distributed system performs a `get` in pure Java mode. Similarly, a `put` made by a C++ client will cause a Java client to hang.

DataSerializable

Using the `DataSerializable` method, the user-defined object is serialized into the following byte array:

```
45 <2-byte-length> <class-name>
```

A Java client can deserialize the byte array, but a C++ client cannot convert the Java class name to a C++ class name.

Implementation

The `DataSerializable` method does not support using a nested object, while `Instantiator.register` does support the use of nested objects. A workaround is to let each Java client manually initiate an object for each possible user object class a C++ client provides, using the following code:

```
User u = new User("", 0);
```

See `Java Serialization Example` for a code sample that shows how to set up user object classes in a Java client.
Using a Custom Class

This example shows how to use the defined BankAccount custom key type and the AccountHistory value type.

The example takes you through these basic operations: registering, creating a cache, connecting to the distributed system, putting data, getting data, and closing the cache.

Using a BankAccount Object

```cpp
#include <gfcpp/GemfireCppCache.hpp>
#include "BankAccount.hpp"
#include "AccountHistory.hpp"
using namespace gemfire;

/*
This example connects, registers types, creates the cache, creates a region, and then puts and gets user defined type BankAccount.
*/
int main( int argc, char** argv )
{
    // Register the user-defined serializable type.
    Serializable::registerType( AccountHistory::createDeserializable );
    Serializable::registerType( BankAccount::createDeserializable );

    CacheFactoryPtr cacheFactory = CacheFactory::createCacheFactory();
    // Create a cache.
    CachePtr cachePtr = cacheFactory->setSubscriptionEnabled(true)
        ->addServer("localhost", 24680)
        ->create();

    // Create a region.
    RegionFactoryPtr regionFactory = cachePtr->createRegionFactory(CACHING_PROXY);
    RegionPtr regionPtr = regionFactory->create("BankAccounts");

    // Place some instances of BankAccount cache region.
    BankAccountPtr KeyPtr(new BankAccount(2309, 123091));
    AccountHistoryPtr ValPtr(new AccountHistory());
    ValPtr->addLog( "Created account" );
    regionPtr->put( KeyPtr, ValPtr );
    printf( "Put an AccountHistory in cache keyed with BankAccount.\n" );
    // Call custom behavior on instance of BankAccount.
    KeyPtr->showAccountIdentifier();
    // Call custom behavior on instance of AccountHistory.
    ValPtr->showAccountHistory();

    // Get a value out of the region.
    AccountHistoryPtr historyPtr =
        dynCast<AccountHistoryPtr>( regionPtr->get( KeyPtr ) );
    if ( historyPtr != NULLPTR ) {
        printf( "Found AccountHistory in the cache.\n" );
        historyPtr->showAccountHistory();
        historyPtr->addLog( "Debit $1,000,000." );
        regionPtr->put( KeyPtr, historyPtr );
        printf( "Updated AccountHistory in the cache.\n" );
    }

    // Look up the history again.
    historyPtr = dynCast<AccountHistoryPtr>( regionPtr->get( KeyPtr ) );
    if ( historyPtr != NULLPTR ) {
        printf( "Found AccountHistory in the cache.\n" );
        historyPtr->showAccountHistory();
    }
    // Close the cache and disconnect from the servers
    cachePtr->close();
    return 0;
}
```
Creating New Statistics

This example provides a programmatic code sample for creating and registering new statistics. For information about the `gemfire_statistics` API, see Statistics API.

Creating New Statistics Programmatically

```cpp
//Get StatisticsFactory
StatisticsFactory* factory = StatisticsFactory::getExistingInstance();

//Define each StatisticDescriptor and put each in an array
StatisticDescriptor** statDescriptorArr = new StatisticDescriptor*[6];
statDescriptorArr[0] = statFactory->createIntCounter("IntCounter",
"Test Statistic Descriptor Int Counter.","TestUnit");
"Test Statistic Descriptor Int Gauge.","TestUnit");
"Test Statistic Descriptor Long Counter.","TestUnit");
"Test Statistic Descriptor Long Gauge.","TestUnit");
"Test Statistic Descriptor Double Counter.","TestUnit");
"Test Statistic Descriptor Double Gauge.","TestUnit");

//Create a StatisticsType
StatisticsType* statsType = statFactory->createType("TestStatsType",
"Statistics for Unit Test.",statDescriptorArr, 6);

//Create Statistics of a given type
Statistics* testStat = factory->createStatistics(statsType,"TestStatistics");

//Statistics are created and registered. Set and increment individual values
Int statIdIntCounter = statsType->nameToId("IntCounter");
testStat->setInt(statIdIntCounter, 10);
testStat->incInt(statIdIntCounter, 1);
int currentValue = testStat->getInt(statIdIntCounter);
```
Part IX

Pivotal GemFire Native Client C# .NET API

*Pivotal GemFire Native Client C# .NET API* describes the primary classes, usage conventions, and C++ to .NET class mappings of the native client .NET API. It demonstrates how to use the API to create caches and perform data serialization.

**About the Native Client C# .NET API**

The Microsoft .NET Framework interface for the Pivotal GemFire native client provides complete access to the native client C++ functionality from any .NET Framework language (C#, C++/CLI, VB.NET, and J#). This enables clients using C# and other .NET languages to use the capabilities provided by the GemFire C++ API.

The GemFire native client uses a set of assemblies managed by the C++ Common Language Infrastructure (C++ CLI). C++ CLI includes the libraries and objects necessary for common language types, and it is the framework for .NET applications.

The .NET API for the native client adds .NET Framework CLI language binding for the GemFire native client product.

Using C#, you can write callbacks and define user objects in the cache. The following figure shows an overview of how a C# application accesses the native client C++ API functionality through C++/CLI.

![Figure 5: C# .NET Application Accessing the C++ API](image)

**Note:** This chapter uses C# as the reference language, but other .NET languages work the same way.

The GemFire C# .NET API is provided in the GemStone::GemFire::Cache::Generic namespace. This namespace allows you to manage your cache, regions, and data using the .NET Generics APIs.
Use the GemFire C# .NET API to programmatically create, populate, and manage a GemFire distributed system.

**Note:** The C# .NET library is thread-safe except where otherwise indicated in the online API documentation.

For complete and current information on the GemFire APIs, see the online .NET API documentation. For general information on .NET, see the Microsoft developer’s network website.
C# .NET Naming and Usage Conventions

Unless noted, the .NET API classes and functions have the same names as their C++ counterparts in the namespace GemStone::GemFire::Cache. In .NET, all method names start with a capital letter.

The .NET interface names match those of comparable C++ interfaces, but with an 'I' prepended to satisfy .NET naming conventions. For example, the .NET equivalent of the C++ CacheLoader interface is ICacheLoader.

The name of the GemFire Serializable interface is IGFSerializable because ISerializable is a .NET built-in type.

Where possible, get* and set* functions are replaced by .NET properties.

You can implement the GemFire .NET interfaces. You cannot extend any of the classes because they are marked as sealed.
Primary APIs in GemStone::GemFire::Cache::Generic

These are the main APIs used for cache, region, and data entry management in GemFire C# .NET. For detailed information about the APIs, see the .NET API documentation included in the product or available online.

**Note:** Declarative configuration via XML of application plug-ins such as cache listener, cache writer, cache loader and partition resolver is not supported when clients are operated in the new .NET Generic API.

**Cache APIs**

This section describes the CacheFactory and Cache classes.

- **CacheFactory class.** Creates a Cache instance based on the provided distributed system and cache configuration. Any gfcpp.properties and cache.xml files provided to the application are used to initialize the system and cache. See Setting System and Cache Properties. If a cache.xml file is used to create a cache and some of the regions already exist, a warning states that the regions exist and the cache is created.

- **Cache class.** Entry point to the GemFire caching API. This class allows you to create regions. The cache is created by calling the create function of the CacheFactory class. When creating a cache, you specify a DistributedSystem that tells the new cache where to find other caches on the network and how to communicate with them.

**Region and Entry APIs**

This section describes classes for working with GemFire regions and region entries.

- **RegionFactory class.** Creates a Region instance based on the provided configuration.

- **IRegion class.** Provides functions for managing regions and cached data. The Region interface implements the generic .NET IDictionary interface. In the GemFire APIs, IRegion implements IDictionary and Region inherits IRegion, giving you access to the full range of .NET Generic collection functions. Use the functions in this class to perform the following actions:
  - Retrieve information about the region, such as its parent region and region attribute objects.
  - Invalidate or destroy the region.
  - Add, update, invalidate, and remove region entries.
  - Determine, individually or as entire sets, the region’s entry keys, entry values, and RegionEntry objects.

- **RegionEntry class.** Contains the key and value for the entry, and provides all non-distributed entry operations. The operations of this class are not distributed and do not affect statistics.

- **RegionShortcut class.** Holds enum definitions for the most common region configurations. Start your region configuration with a shortcut setting in the region attribute, refid. Then customize further using the RegionAttributes.

- **RegionAttributes class.** Holds all attribute values for a region and provides functions for retrieving all attribute settings. This class can only be modified by the AttributesFactory class before region creation, and the AttributesMutator class after region creation.

- **AttributesMutator class.** Allows modification of an existing region’s attributes for application plug-ins and expiration actions. Each region has an AttributesMutator instance.

**Data Serialization APIs**

Use either IPdxSerializable or IGFSerializable for each region. Do not mix the two.

For more information on these options, see Data Serialization.
- **IPdxSerializable interface.** Provides a flexible way to serialize your domain objects for cache storage and transfer to the servers. This is a GemFire built-in serialization framework. See [Data Serialization](#).
- **IPdxReader.** Supplies operations for reading data from GemFire IPDXSerializable types.
- **IPdxWriter.** Provides operations for writing data into GemFire IPDXSerializable types.
- **IPdxInstance.** Instance of a PDX serialized object that you can use to access the object’s data without having to deserialze the object first.
- **IPdxInstanceFactory.** Allows you to build an IPdxInstance using raw data.
- **IPdxTypeMapper interface.** Allows you to map .NET type names to Java type names when using PDX serialization.
- **IGFSerializable interface.** Superclass of one set of user objects that can be serialized and stored in the cache. These are GemFire built-in serializable types. See [Data Serialization](#).
- **Serializable class.** Wraps the native C++ `gemfire::Serializable` objects as managed `IGFSerializable` objects. Whenever native C++ clients and .NET clients interoperate and are part of the same distributed system, the user-defined types that are put by the native C++ clients that have not been defined in .NET are returned as objects of this class.

The API contains overloads for most Region methods and other methods that take `Serializable` as a value and that are more optimized than the more generic `IGFSerializable` overloads. The application prefers using these overloads whenever the base class of an object is `Serializable`.

- **DataInput.** Supplies operations for reading primitive data values and user-defined objects from a byte stream.
- **DataOutput.** Provides operations for writing primitive data values and user-defined objects implementing `IGFSerializable` to an integer.

### Event Handling APIs

Code your event handlers to do minimal work before returning control to GemFire.

For example, a listener implementation may hand off the event to a thread pool that processes the event on its thread rather than the listener thread. Exceptions thrown by the listeners are caught by GemFire and logged.

- **RegionEvent class.** Provides information about the event, such as what region the event originated in, whether the event originated in a cache remote to the event handler, and whether the event resulted from a distributed operation.
- **EntryEvent class.** Provides all available information for the `RegionEvent`. It also provides entry-specific information, such as the old and new entry values and whether the event resulted from a load operation.
- **ICacheLoader application callback interface.** Loads data into a region.
- **ICacheWriter application callback interface.** Synchronously handles region and entry events before the events occur. Entry events are `create`, `update`, `invalidate`, and `destroy`. Region events are `invalidate` and `destroy`. This class has the ability to abort events.
- **ICacheListener application callback interface.** Asynchronously handles region and entry events. Listeners receive notifications when entries in a region change, or when changes occur to the region attributes themselves. Entry events are `create`, `update`, `invalidate`, and `destroy`. Region events are `invalidate` and `destroy`. Multiple events can cause concurrent invocation of `ICacheListener` methods. If event A occurs before event B, there is no guarantee that their corresponding `ICacheListener` method invocations will occur in the same order.

### Property Collections and Logging APIs

This section describes classes for property collections and logging.

- **Properties class.** Provides a collection of properties, each of which is a key/value pair. Each key is a string, and the value can be a string or an integer.
• **Log class.** Defines methods available to clients that need to write a log message to their GemFire system shared log file. Any attempt to use an instance after its connection is disconnected throws a **NotConnectedException.** For any logged message the log file contains:
  - The log level of the message.
  - The time the message was logged.
  - The ID of the connection and thread that logged the message.
  - The message itself, possibly with an exception including its stack trace.

## C++ Class to .NET Class Mappings

Wherever the native C++ class methods use pass-by-reference semantics to return data, the corresponding .NET methods return the object instead of using pass-by-reference semantics.

### Table 16: C++ Class to .NET Class Mappings

<table>
<thead>
<tr>
<th>C++ Class</th>
<th>.NET Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>class gemfire::AttributesFactory</td>
<td>Sealed class AttributesFactory</td>
</tr>
<tr>
<td>class gemfire::AttributesMutator</td>
<td>Sealed class AttributesMutator</td>
</tr>
<tr>
<td>class gemfire::Cache</td>
<td>Sealed class Cache</td>
</tr>
<tr>
<td>abstract class gemfire:: Cacheable</td>
<td>Interface IPdxSerializable or interface IGFSerializable</td>
</tr>
<tr>
<td>class gemfire::CacheableBytes</td>
<td>Byte[] or ArrayList&lt;Byte&gt;</td>
</tr>
<tr>
<td>class gemfire::CacheableInt32</td>
<td>Int32</td>
</tr>
<tr>
<td>class gemfire::CacheableString</td>
<td>String</td>
</tr>
<tr>
<td>abstract class gemfire:: CacheableKey</td>
<td>You can use any type that implements hashcode and equals. The generic .NET built-in types are all suitable.</td>
</tr>
<tr>
<td>abstract class gemfire::CacheListener</td>
<td>Interface ICacheListener</td>
</tr>
<tr>
<td>class gemfire:: CacheLoader</td>
<td>Interface ICacheLoader plus static class CacheLoader</td>
</tr>
<tr>
<td>class gemfire::CacheWriter</td>
<td>Interface class ICacheWriter</td>
</tr>
<tr>
<td>class gemfire::CacheFactory</td>
<td>Sealed class CacheFactory</td>
</tr>
<tr>
<td>class gemfire::DataInput</td>
<td>With IPdxSerializable, IPdxReader.</td>
</tr>
<tr>
<td></td>
<td>With IGFSerializable, sealed class DataInput.</td>
</tr>
<tr>
<td>class gemfire:: DataOutput</td>
<td>With IPdxSerializable, IPdxWriter.</td>
</tr>
<tr>
<td></td>
<td>With IGFSerializable, sealed class DataOutput.</td>
</tr>
<tr>
<td>C++ Class</td>
<td>.NET Class</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>class gemfire:: DiskPolicyType</td>
<td>enum DiskPolicyType plus static class DiskPolicy containing convenience methods for DiskPolicyType enumeration</td>
</tr>
<tr>
<td>class gemfire:: DistributedSystem</td>
<td>Sealed class DistributedSystem</td>
</tr>
<tr>
<td>class gemfire:: EntryEvent</td>
<td>Sealed class EntryEvent</td>
</tr>
<tr>
<td>class gemfire:: Exception</td>
<td>Class GemfireException</td>
</tr>
<tr>
<td>all other exceptions deriving from gemfire:: Exception</td>
<td>Corresponding exceptions deriving from GemfireException</td>
</tr>
<tr>
<td>class gemfire:: ExpirationAction</td>
<td>enum ExpirationAction plus static class Expiration containing convenience methods for ExpirationAction enumeration</td>
</tr>
<tr>
<td>class gemfire:: Log</td>
<td>Static class Log. The native Log::log method is mapped to Log.Write to avoid the conflict with the class name which is reserved for the constructors of Log class. The various loglevel Throw or Catch methods are not implemented, since they are redundant to Log::Log, Log::LogThrow, and Log::LogCatch methods that take LogLevel as a parameter.</td>
</tr>
<tr>
<td>enum gemfire:: MemberType</td>
<td>enum MemberType</td>
</tr>
<tr>
<td>abstract class gemfire:: PersistanceManager</td>
<td>Not provided. You can register a C++ implementation using AttributesFactory. SetPersistenceManager but you cannot implement a new one in .NET</td>
</tr>
<tr>
<td>class gemfire:: Properties</td>
<td>Sealed class Properties</td>
</tr>
<tr>
<td>class gemfire:: Properties::Visitor</td>
<td>Delegate PropertiesVisitor</td>
</tr>
<tr>
<td>abstract class gemfire:: Region</td>
<td>Class IRegion</td>
</tr>
<tr>
<td>class gemfire:: RegionAttributes</td>
<td>Sealed class RegionAttributes</td>
</tr>
<tr>
<td>class gemfire:: ScopeType</td>
<td>enum ScopeType plus static class Scope containing convenience methods for ScopeType enumeration</td>
</tr>
</tbody>
</table>
Java to .NET Type Mapping Table

The following table provides a mapping between Java and .NET types.

**Table 17: Java types and .NET types**

<table>
<thead>
<tr>
<th>Java Type</th>
<th>.NET Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>instances of <code>PdxSerializable</code></td>
<td>.NET class of same name</td>
</tr>
<tr>
<td>instances of <code>PdxInstance</code></td>
<td>.NET class of same name</td>
</tr>
<tr>
<td>instances serialized by a <code>PdxSerializer</code></td>
<td>.NET class of same name</td>
</tr>
<tr>
<td><code>java.lang.Byte</code></td>
<td><code>System.SByte</code></td>
</tr>
<tr>
<td><code>java.lang.Boolean</code></td>
<td><code>System.Boolean</code></td>
</tr>
<tr>
<td><code>java.lang.Character</code></td>
<td><code>System.Char</code></td>
</tr>
<tr>
<td><code>java.lang.Short</code></td>
<td><code>System.Int16</code></td>
</tr>
<tr>
<td><code>java.lang.Integer</code></td>
<td><code>System.Int32</code></td>
</tr>
<tr>
<td><code>java.lang.Long</code></td>
<td><code>System.Int64</code></td>
</tr>
<tr>
<td><code>java.lang.Float</code></td>
<td><code>System.Float</code></td>
</tr>
<tr>
<td><code>java.lang.Double</code></td>
<td><code>System.Double</code></td>
</tr>
</tbody>
</table>
Object Lifetimes

The .NET API provides a managed set of assemblies for the C++ API. The underlying C++ object will stay in memory until the .NET object is garbage-collected.

The underlying C++ API employs reference counting using smart pointers for most classes. This means that all API operations with those objects return a reference to the underlying object and not a copy. Consequently, the underlying object will not be freed as long as the .NET application holds a reference to an object. In other words, the underlying object will stay in memory until the .NET object is garbage-collected. As long as a reference to an object is alive, the artifacts it maintains will also be alive.
For example, as long as a Region object is not garbage-collected, then the destructor of the C++ native persistence manager (if any) for the region is not invoked.

In the C++ API, the references to an object are reduced when the object goes out of scope for stack allocation, or is deleted explicitly for heap allocation. The object is destroyed when its reference count reaches zero. In the .NET API, the references are reduced when the object is garbage-collected or is explicitly disposed with the .NET using statement.

Because a reference to the objects is returned, any change to the object also immediately changes the object as stored internally. For example, if an object is put into the cache using Region.Put, a reference of the object is stored in the internal structures. If you modify the object, the internal object also changes. However, it is not distributed to other members of the distributed system until another Region.Put is done.

To find out if a class is reference counted, look at the online API documentation for the class. If the class is wrapped by UMWrap or SBWrap, the class is reference counted.

These are examples of classes that are reference counted:

- Cache
- CacheStatistics
- DistributedSystem
- Properties
- RegionAttributes
- AttributesMutator
- RegionEntry
- Region
- EntryEvent
- RegionEvent

These are examples of classes that are not reference counted:

- AttributesFactory
- DataInput
- DataOutput
- SystemProperties

**.NET Application Domains**

Application domains, or AppDomains, are units of isolation, security boundaries, and loading and unloading for applications in the .NET runtime. Multiple application domains can run in a single process. Each can have one or many threads, and a thread can switch application domains at runtime.

**Note:** Before you use application domains with GemFire, make sure you understand how to use them generally in your .NET programming.

The .NET managed assemblies require interface methods invoked by the native C++ layer to be in the same AppDomain as that of the .NET DLL. If not, an exception is thrown because the thread is unable to cross AppDomain boundaries.
Problem Scenarios

These scenarios describe processes and implementations that should be avoided when using AppDomains.

- **Using Application Callbacks**
- **Loading an Application DLL in Multiple AppDomains**
- **Native Client inside IIS**

### Using Application Callbacks

**Scenario:** A .NET thread loads the GemFire DLL in application domain AD1. This thread may have access to the other domains in the application if code access security allows it. This thread can then call `AppDomain.CreateInstance` to create a callback object (ICacheListener, ICacheLoader, or ICacheWriter) in another domain called AD2. If the callback object is marshaled by reference, the callback is executed in the domain where it is created (AD2). The thread that loads the GemFire DLL in domain AD1 runs the callback methods in the second domain, AD2. An exception is thrown when the callback method is invoked because the native code that invokes the callback is not allowed to cross the AppDomain boundary.

**Resolution:** When an application creates and unloads application domains it should ensure that the application domain where the GemFire .NET DLL is loaded is the same domain where the application callback and IGFSerializable objects are created.

### Loading an Application DLL in Multiple AppDomains

**Scenario:** the application loads the GemFire DLL in one application domain, then reloads the GemFire DLL in another application domain (with or without unloading the previous AppDomain). The callbacks, as well as other interface implementations, like IPdxSerializable and IGFSerializable, throw exceptions because the native C++ code does not know about AppDomains and is loaded only once in the initial AppDomain.

**Resolution:** The application should always use the first AppDomain to load the GemFire DLL, or it should not load the GemFire DLL multiple times.

### Native Client inside IIS

**Scenario:** When you deploy more than one web application inside an Internet Information Service (IIS), the IIS creates an appdomain subprocess for each web application in the single process, but the native client C++ cache instance remains a singleton in the process. Because of this, you can run into conflicts between cache creation and closure by the different appdomains. For example, if one appdomain calls `cache.close`, it closes the cache for the entire process. Any further cache access operations by the other appdomains return cache closed exceptions.

**Resolution:** Cache create/close provides reference counting of Cache create and close. Each process can use the counter to make sure it creates the Cache once and closes it once. To enable this, set the GemFire system property, `appdomain-enabled` to true.
Creating a Cache

You create a cache using the GemFire `CacheFactory.Create` call. Cache creation initializes the distributed system and creates the cache using your `gfcpp.properties` and `cache.xml` file settings and any additional properties you provide to the call.

See Setting System and Cache Properties and see Cache Initialization File.

Connecting and Creating the Cache

In this example, the application connects to the distributed system and creates the cache using the available configuration files.

The application becomes a distributed system member in the cache Create call.

```csharp
CacheFactory cacheFactory = CacheFactory.CreateCacheFactory();
Cache cache = cacheFactory.Create();
```

Providing Properties to the Cache Creation

You can also create a cache by referencing a `cache.xml` file, as shown in the following example. You can use the `Properties` object to change any of the `gfcpp.properties` settings.

```csharp
Properties prop = Properties.Create();
prop.Insert("cache-xml-file", "cache.xml");
CacheFactory cacheFactory = CacheFactory.CreateCacheFactory(prop);
Cache cache = cacheFactory.Create();
```

For systems with security enabled, the credentials for a joining member are authenticated when the cache is created and the system connection is made. For more information about secure connections to a distributed system, see Security.

Creating a Region

To create a region, you create a `RegionFactory` using the `RegionShortcut` that most closely fits your region configuration.

From that point, you customize the settings for region attributes as needed.

Creating a region using the native client API only creates a proxy client side region. A corresponding region with the same name and path must also exist on the servers that have been configured for client connections and upon which the client will perform its operations.

Creating a CACHING_PROXY Region

This example creates a region using a CACHING_PROXY RegionShortcut with no further modifications:

```csharp
RegionFactory regionFactory = cache.CreateRegionFactory(RegionShortcut.CACHING_PROXY);
IRegion<string, string> region = regionFactory.Create<string, string>("exampleRegion");
```
Adding an Entry to the Cache

You populate a native client region with cache entries by using the generic IDictionary API or by using the .NET Region.Put or the Region.Create API functions.

The Put function places a new value into a region entry with the specified key, while the Create function creates a new entry in the region. The Put and Create functions provide a user-defined parameter object to any cache writer invoked in the process.

If a value for the entry key already exists in the cache when you add an entry, GemFire overwrites the previously cached value. New values in the cache are propagated to the connected cache server.

The .NET Generics provide type safety, so you cannot change your entry key and value types once you have begun to populate the region. If you need to use different types for the same region, store them all inside objects in the region.

Using the API to Put Values Into the Cache

In this example, the program puts entries into the cache with string values.

Batch Operations: Using PutAll to Add Multiple Entries

You can batch up multiple key/value pairs into a hashmap and put them into the cache with a single operation using the .NET Region.PutAll API function. Each entry is processed for interest registration on the server, so each entry requires its own unique event ID. Updates and creates can be mixed in a PutAll operation, so those events need to be addressed on the cache server for appropriate cache listener invocation on distributed system members. Map entries retain their original order when they are processed at the server.
Accessing an Entry

The region entry retrieval methods return the value associated with the specified key, and pass the callback argument to any cache loaders or cache writers that are invoked during the operation.

If the value is not available locally, it is requested from the server. If the server request is unsuccessful, a local cache loader is invoked, if one is available. The operation throws `keyNotFoundException` if the Region is unable to retrieve a value through any of these means.

### Using the Region API to Retrieve Values From the Cache

Here, the program retrieves two entries from the region.

```java
string value1 = region1["Key1"];
string value2 = region1["Key2"];  
int valueA = region2["KeyA"];  
int valueB = region2["KeyB"];  
string valueQ = region.Get(111, null);
string valueR = region.Get(222, null);
```

Batch Operations: Using `getAll` to Return Values from Multiple Entries

The `GetAll` region API returns values for collection of keys from the local cache or server.

If value for a key is not present locally, then it is requested from the Java server. The value returned is not copied, so multi-threaded applications should not modify the value directly but should use the update methods.

This method is not applicable to local region instances.


Removing an Entry

The standard `Region::Remove` API removes the entry with specified key and provides a user-defined parameter object to any `CacheWriter` or `CacheListener` invoked in the process.

The `Remove` call not only removes the value, but also the key and entry from this region. The remove operation is propagated to the GemFire cache server that the native client is connected to. If the destroy operation fails due to an exception on the server (for example, a `CacheServerException` or security exception), then the local entry is still removed.

The `Remove` operation updates `CacheStatistics::LastAccessedTime`, `CacheStatistics::HitCount`, and `CacheStatistics::MissCount` for this region and the entry.

The `Remove` API returns true if the entry (key, value) has been removed or false if the entry (key, value) has not been removed.

**Bulk Remove Operations Using `removeAll`**

You can use the `Region::removeAll` API to remove all entries for a collection of specified keys from the region. The effect of this call is equivalent to that of calling `destroy` on this region once for
each key in the specified collection. If an entry does not exist, then that key is skipped. Note that an
\texttt{EntryNotFoundException} is not thrown.

The \texttt{RemoveAll} operation updates \texttt{CacheStatistics::LastAccessedTime},
\texttt{CacheStatistics::HitCount}, and \texttt{CacheStatistics::MissCount} for this region and the entries that are
removed.

The \texttt{RemoveAll} API also supports providing a callback argument to any cache loaders or cache writers that
are invoked in the operation. See the Region online API documentation for more information about using
\texttt{RemoveAll}.

\section*{Data Serialization}

All data that GemFire moves out of the local cache must be serializable.

Region data that must be serializable falls under the following categories:

• Partitioned regions (except functions that add data locally to a partitioned region use the deserialized
  form).
• Distributed regions.
• Regions that are persisted or overflowed to disk.
• Server or client regions in a client/server installation.
• Regions distributed between gateways in a multi-site installation.
• Regions that receive events from remote caches.
• Regions that provide function arguments and results.

To minimize the cost of serialization and deserialization, GemFire avoids changing the data format
whenever possible. This means your data may be stored in the cache in serialized or deserialized form,
depending on how you use it. For example, if a server acts only as a storage location for data distribution
between clients, it makes sense to leave the data in serialized form, ready to be transmitted to clients that
request it. Partitioned region data is always stored in serialized form with one exception—functions that
add data to a partitioned region locally use the deserialized form.
Data Serialization Options

Built-in .NET types are serialized automatically into the cache and can be retrieved by Java servers and other GemFire clients. For domain objects that are not simple types, you have three GemFire serialization options.

The options give good performance and flexibility for data storage, transfers, and language types. The GemFire options can also improve performance in serializing and deserializing built-in types.

The simplest option is to use perform automatic serialization by registering the GemFire .NET PDX reflection-based autoserializer in your application. When you have this registered, GemFire uses it for all domain objects that are not custom serialized.

You can also custom serialize your objects by implementing one of the GemFire .NET interfaces, `GemStone::GemFire::Cache::Generic::IPdxSerializable` or `GemStone::GemFire::Cache::IGFSerializable`.

You also have the option of using default .NET serialization, but you cannot use it unless you also use helper classes. The helper classes you must use are `CacheableObject` and `CacheableObjectXml`.

GemFire .NET PDX serialization has more bytes in overhead than GemFire .NET Data serialization, but using PDX serialization helps you avoid the performance costs of deserialization when performing queries. Applications can use `PdxInstances` in functions to avoid the deserialization of entire objects.

Table 18: Serialization Options: Comparison of Features

<table>
<thead>
<tr>
<th>Capability</th>
<th>IGFSerializable</th>
<th>IPdxSerializable and PDX reflection-based autoserializer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handles multiple versions of domain objects*</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Provides single field access on servers of serialized data, without full deserialization. Supported also for OQL queries.</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Automatically ported to other languages by GemFire - no need to program Java-side implementation</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Works with GemFire delta propagation</td>
<td>X</td>
<td>X (See explanation below.)</td>
</tr>
</tbody>
</table>

*You can mix domain object versions where the differences between versions are the addition and removal of object fields.

By default, you can use GemFire delta propagation with PDX serialization. However, delta propagation will not work if you have set the GemFire property read-serialized to "true". In terms of deserialization, to apply a change delta propagation requires a domain class instance and the `fromDelta` method. If you have set read-serialized to true, you will receive an `IPdxInstance` instead of a domain class instance and `IPdxInstance` does not have the `fromDelta` method required for delta propagation. You will also require the Java domain class on the server similar to the you would need the .NET PDX Delta domain class.

For detailed information on the interfaces, see the online API documentation.
Serialize with GemFire PDX Serialization

Pivotal GemFire's Portable Data eXchange (PDX) is a cross-language data format that can reduce the cost of distributing and serializing your objects. PDX stores data in named fields that you can access individually, to avoid the cost of deserializing the entire data object. PDX also allows you to mix versions of objects where you have added or removed fields.

PDX serialization is supported by C++ clients. If you have C++ clients that may receive events for a region, you can now use PDX serialization on the region's entries.

You have two options for GemFire PDX serialization when using the C# .NET caching API. You can program your domain objects using the IPdxSerializable interface, or you can use GemFire's reflection-based autoserializer.

GemFire PDX Serialization Features

GemFire PDX serialization offers several advantages.

Application Versioning of PDX Domain Objects

Domain objects evolve along with your application code. You may create an address object with two address lines, then realize later that a third line is required for some situations. Or you may realize that a particular field is not used and want to get rid of it.

With PDX, you can use old and new versions of domain objects together in a distributed system if the versions differ by the addition or removal of fields. This compatibility lets you gradually introduce modified code and data into the system, without bringing the system down.

Pivotal GemFire maintains a central registry of the PDX domain object metadata. Using the registry, GemFire preserves fields in each member's cache regardless of whether the member has the field defined. When a member receives an object that has a field registered that the member is not aware of, the member does not access the field, but preserves it and passes it along with the rest of the object to other members. When a member receives an object that is missing one or more fields according to the member's version, GemFire assigns the .NET default values for the field types to the missing fields.

Portability of PDX Serializable Objects

When you create an IPdxSerializable object, GemFire stores the object's type information in a central registry. The information is passed between peers, between clients and servers, and between distributed systems.

This offers a notable advantage to the .NET client, which shares data with Java cache servers. Clients automatically pass registry information to servers when they store an IPdxSerializable object. Clients can run queries and functions against the data in the servers without the servers needing to know anything about the stored objects. One client can store data on the server to be retrieved by another client, with the server never needing to know the object type. This means you can code your .NET clients to manage data using Java servers without having to create Java implementations of your .NET domain objects.

Reduced Deserialization of Serialized Objects

The access methods for IPdxSerializable Objects allow you to examine specific fields of your domain object without deserializing the entire object. This can reduce serialization and deserialization costs significantly. .NET clients can run queries and execute functions against the objects in the server caches without deserializing the entire object on the server side. The query engine automatically recognizes PDX objects and uses only the fields it needs.

Clients can execute Java functions on server data that only access parts of the domain objects by using PdxInstance.
Likewise, peers can access just the fields needed from the serialized object, keeping the object stored in the cache in serialized form.

**Serialize Using the GemFire PDX Autoserializer**

When you register the reflection-based serializer, GemFire uses it to serialize all objects that do not implement IPdxSerializable or IGFSerializable. You can customize the auto-serialization behavior for your domain objects by adding serialization attributes to your object’s fields.

**Procedure**

1. If you have not already registered the PDX reflection-based autoserializer, add the registration code to your application.
   
   For example:
   ```csharp
   ...
   // Register reflection-based autoserializer to serialize
   // domain objects using PDX serialization
   Serializable.RegisterPdxSerializer(new ReflectionBasedAutoSerializer());
   ```
   
   This can only be configured in the application code. It cannot be configured declaratively in cache.xml.

2. (Optional) For each object you intend to have autoserialized, customize the serialization as needed.
   
   **Note:** If you also use PDX serialization in Java for the object, customize your serialization the same for both languages.

   a. The following extension methods apply to autoserialization:
      
      • **WriteTransform.** Controls what field value is written during auto serialization.
      • **ReadTransform.** Controls what field value is read during auto deserialization.
      • **GetFieldTransform.** Defines the specific field names that will be generated during autoserialization.
      • **IsIdentityField.** Controls which field is marked as the identity field. Identity fields are used when a PdxInstance computes its hash code to determine whether it is equal to another object.
      • **GetFieldTransform.** Determines the field type that will be used when autoserializing the given field.
      • **IsFieldIncluded.** Specifies which fields of a class to autoserialize.

   See [Extending the Autoserializer](Extending-the-Autoserializer) for sample usage.

   b. If you are writing a Java application, you can use the IPdxTypeMapper to map Java types to .NET types. Note that you only need to use the IPdxTypeMapper if you are writing Java applications.

   See [Map .NET Domain Type Names to PDX Type Names with IPdxTypeMapper](Map-.NET-Domain-Type-Names-to-PDX-Type-Names-with-IPdxTypeMapper) for sample usage.

   c. To specify an identifier field in your domain object, add the attribute PdxIdentityField to the field.

   For example:
   ```csharp
   [PdxIdentityField] private int id;
   ```

   d. To exclude a field from serialization, add the .NET attribute NonSerialized to the field.

   For example:
   ```csharp
   [NonSerialized] private int myLocalData;
   ```

   For each domain class GemFire serializes using the autoserializer, all fields are considered for serialization except those defined as static, literal or readonly and those you explicitly exclude using the .NET NonSerialized attribute.
**Extend the PDX Autoserializer**

This example code demonstrates how to extend the autoserializer to customize serialization.

```csharp
public class AutoSerializerEx : ReflectionBasedAutoSerializer
{
    public override object WriteTransform(FieldInfo fi, Type type, object originalValue)
    {
        if (fi.FieldType.Equals(Type.GetType("System.Guid")))
        {
            return originalValue.ToString();
        }
        else if (fi.FieldType.Equals(Type.GetType("System.Decimal")))
        {
            return originalValue.ToString();
        }
        else
        {
            return base.WriteTransform(fi, type, originalValue);
        }
    }
    public override object ReadTransform(FieldInfo fi, Type type, object serializeValue)
    {
        if (fi.FieldType.Equals(Type.GetType("System.Guid")))
        {
            Guid g = new Guid((string)serializeValue);
            return g;
        }
        else if (fi.FieldType.Equals(Type.GetType("System.Decimal")))
        {
            return Convert.ToDecimal((string)serializeValue);
        }
        else
        {
            return base.ReadTransform(fi, type, serializeValue);
        }
    }
    public override FieldType GetFieldType(FieldInfo fi, Type type)
    {
        if (fi.FieldType.Equals(Type.GetType("System.Guid")) || fi.FieldType.Equals(Type.GetType("System.Decimal")))
        {
            return FieldType.STRING;
        }
        return base.GetFieldType(fi, type);
    }
    public override bool IsIdentityField(FieldInfo fi, Type type)
    {
        if (fi.Name == "_identityField")
        {
            return true;
        }
        return base.IsIdentityField(fi, type);
    }
    public override string GetFieldName(FieldInfo fi, Type type)
    {
        if (fi.Name == "_nameChange")
        {
            return fi.Name + "NewName";
        }
        return fi.Name;
    }
    public override bool IsFieldIncluded(FieldInfo fi, Type type)
    {
        if (fi.Name == "_notInclude")
        {
            return false;
        }
        return base.IsFieldIncluded(fi, type);
    }
}
```
Serialize Your Domain Objects with IPdxSerializer

For domain objects that you cannot or do not want to modify, use the IPdxSerializer class to serialize and deserialze the object's fields.

You use one IPdxSerializer implementation for the entire cache, programming it for all of the domain objects that you handle in this way. This way you do not have to implement the IPdxSerializable interface for each domain class.

With IPdxSerializer, you leave your domain object as-is and handle the serialization and deserialization in the separate serializer. You register the serializer in your cache PDX configuration. Then program the serializer to handle all of the domain objects you need.

If you write your own IPdxSerializer and you also use the ReflectionBasedAutoSerializer, then the IPdxSerializer needs to own the ReflectionBasedAutoSerializer and delegate to it. A cache can only have a single IPdxSerializer instance.

Note: The IPdxSerializer toData and fromData methods differ from those for IPdxSerializable. They have different parameters and results.

To register an IPdxSerializer, you can use the following code. Note that you can only register the IPdxSerializer in the application code. It cannot be configured declaratively in cache.xml.

Example:

```csharp
...
// Register a PdxSerializer to serialize
// domain objects using PDX serialization
Serializable.RegisterPdxSerializer(new MyPdxSerializer());
```

Serialize Using the GemFire IPdxSerializable Interface

Use this procedure to program your domain object for PDX serialization using the IPdxSerializable Interface. When you write objects using PDX serialization, they are distributed to the server tier in PDX serialized form. When you run queries against the objects on the servers, only the fields you specify are deserialized.

Procedure

1. In your domain class, implement GemStone::GemFire::Cache::Generic::IPdxSerializable. Example:

```csharp
...
public class PortfolioPdx : IPdxSerializable
```

2. If your domain class does not have a zero-arg constructor, create one for it.

3. Program the IPdxSerializable ToData function to serialize your object as required by your application.

   a. Write your domain class's standard. NET data fields using the IPdxWriter write methods. GemFire automatically provides IPdxWriter to the ToData function for IPdxSerializable objects.
   
   b. Call the ToData markIdentifyField function for each field GemFire should use to identify your object. This is used to compare objects for operations like DISTINCT queries. The markIdentifyField call must come after the associated field write methods.
Example:

```csharp
// object fields
private int m_id;
private string m_pkid;
private PositionPdx m_position1;
private PositionPdx m_position2;
private Hashtable m_positions;
private string m_type;
private string m_status;
private string[] m_names;
private byte[] m_newVal;
private DateTime m_creationDate;
private byte[] m_arrayZeroSize;
private byte[] m_arrayNull;

// ToData
public void ToData(IPdxWriter writer)
{
    writer.WriteInt("id", m_id);
    // identity field
    .MarkIdentityField("id")
    .WriteString("pkid", m_pkid)
    .WriteObject("position1", m_position1)
    .WriteObject("position2", m_position2)
    .WriteObject("positions", m_positions)
    .WriteString("type", m_type)
    .WriteString("status", m_status)
    .WriteByteArray("newVal", m_newVal)
    .WriteDate("creationDate", m_creationDate)
    .WriteByteArray("arrayNull", m_arrayNull)
    .WriteByteArray("arrayZeroSize", m_arrayZeroSize);
}
```

4. Program IPdxSerializable FromData to read your data fields from the serialized form into the object's fields using the IPdxReader read methods. GemFire automatically provides IPdxReader to the FromData function for IPdxSerializable objects.

Use the same names as you did in ToData and call the read operations in the same order as you called the write operations in your ToData implementation.

Example:

```csharp
public void FromData(IPdxReader reader)
{
    m_id = reader.ReadInt("id");
    bool isIdentity = reader.IsIdentityField("id");

    if (isIdentity == false)
        throw new IllegalStateException("Portfolio id is identity field");

    bool isId = reader.HasField("id");

    if (isId == false)
        throw new IllegalStateException("Portfolio id field not found");

    bool isNotId = reader.HasField("ID");

    if (isNotId == true)
        throw new IllegalStateException("Portfolio isNotId field found");

    m_pkid = reader.ReadString("pkid");
    m_position1 = (PositionPdx)reader.ReadObject("position1");
    m_position2 = (PositionPdx)reader.ReadObject("position2");
    m_positions = (Hashtable)reader.ReadObject("positions");
    m_type = reader.ReadString("type");
    m_status = reader.ReadString("status");
    m_names = reader.ReadStringArray("names");
```
5. Optionally, program your domain object’s equals and hashcode methods.

**Program Your Application to Use IPdxInstance**

An IPdxInstance is a lightweight wrapper around PDX serialized bytes. It provides applications with runtime access to fields of a PDX serialized object.

You can configure your cache to return an IPdxInstance when a PDX serialized object is deserialized instead of deserializing the object to a domain class. You can then program your application code that reads your entries to handle IPdxInstances fetched from the cache.

**Note:** This option applies only to entry retrieval that you explicitly code using methods like EntryEvent.getNewValue and Region.get, as you do inside functions or in cache listener code. This does not apply to querying because the query engine retrieves the entries and handles object access for you.

**Note:** IPdxInstance overrides any custom implementation you might have coded for your object’s equals and hashcode methods.

**Procedure**

1. In the cache.xml file of the server member where entry fetches are run, set the <pdx> read-serialized attribute to true.

   Data is not necessarily accessed on the memer that you have coded for it. For example, if a client application runs a function on a server, the actual data access is done on the server, so you set read-serialized to true on the server.

   For example:

   ```
   // Cache configuration setting PDX read behavior
   <cache>
      <pdx read-serialized="true" />  
   ... </cache>
   ```

2. Write the application code that fetches data from the cache to handle a IPdxInstance. If you are sure you will only retrieve IPdxInstances from the cache, you can code only for that. In many cases, a IPdxInstance or a domain object may be returned from your cache entry retrieval operation, so you should check the object type and handle each possible type.

   **See** Creating an IPdxInstance with IPdxInstanceFactory for an example of this.

   If you configure your cache to allow PDX serialized reads, cache fetches return the data in the form it is found. If the object is not serialized, the fetch returns the domain object. If the object is serialized, the fetch returns the PdxInstance for the object.

   **Note:** If you are using IPdxInstances, you cannot use delta propagation to apply changes to PDX serialized objects.

   For example, in client/server applications that are programmed and configured to handle all data activity from the client, PDX serialized reads done on the server side will always return the IPdxInstance. This is because all of data is serialized for transfer from the client and you are not performing any server-side activities that would deserialize the objects in the server cache.

   In mixed situations, such as where a server cache is populated from client operations and also from data loads done on the server side, fetches done on the server can return a mix of IPdxInstances and domain objects.
When fetching data in a cache with PDX serialized reads enabled, the safest approach is to code to handle both types, receiving an Object from the fetch operation, checking the type and casting as appropriate.

**Use the IPdxInstanceFactory to Create IPdxInstances**

You can use the IPdxInstanceFactory to create an IPdxInstance from raw data when the domain class is not available on the server.

This option can be useful when you need an instance of a domain class for plug-in code such as a function or a loader. If you have the raw data for the domain object (the class name and each field’s type and data), then you can explicitly create a IPdxInstance. The IPdxInstanceFactory is very similar to the IPdxWriter except that after writing each field, you need to call the create method which returns the created IPdxInstance.

### Creating an IPdxInstance with IPdxInstanceFactory

```csharp
/*
 * The PdxInstance QuickStart Example.
 * This example takes the following steps:
 *  * This example shows IPdxInstanceFactory and IPdxInstance usage.
 *  * 1. Create a GemFire Cache.
 * 2. Creates the PdxInstanceFactory for Person class.
 * 3. Then creates instance of PdxInstance
 * 4. It does put.
 * 5. Then it does get and access it fields.
 */

// Use standard namespaces
using System;
using System.Reflection;

// Use the GemFire namespace
{
    public class Person
    {
        private string name;
        //this is the only field used on server to create hashcode
        and use in equals method
        [PdxIdentityField]
        private int id;
        private int age;
        public Person() { }
        public Person(string name, int id, int age)
        {
            this.name = name;
            this.id = id;
            this.age = age;
        }
        #region Public Properties
        public string Name
        {
            get { return name; }
        }
        public int ID
        {
            get { return id; }
        }
        public int Age
        {
            get { return age; }
        }
        #endregion
```
class PdxInstance
{
    static void Main(string[] args)
    {
        try {
            CacheFactory cacheFactory = CacheFactory.CreateCacheFactory(null);
            Console.WriteLine("Connected to the GemFire Distributed System");
            // Create a GemFire Cache with the "clientPdxRemoteQuery.xml" Cache XML file.
            // Set SetPdxReadSerialized to true to access PdxInstance Cache cache = cacheFactory.Set("cache-xml-file", "XMLs/clientPdxInstance.xml").Create();
            Console.WriteLine("Created the GemFire Cache");
            // Get the example Region from the Cache which is declared in the Cache XML file.
            IRegion<string, IPdxInstance> region = cache.GetRegion<string, IPdxInstance>("Person");
            Console.WriteLine("Obtained the Region from the Cache");
            Person p = new Person("Jack", 7, 21);
            //PdxInstanceFactory for Person class
            IPdxInstanceFactory pif = cache.CreatePdxInstanceFactory("Person");
            pif.WriteString("name", p.Name);
            pif.WriteInt("id", p.ID);
            pif.MarkIdentityField("id");
            pif.WriteInt("age", p.Age);
            IPdxInstance pdxInstance = pif.Create();
            Console.WriteLine("Created PdxInstance for Person class");
            region["Key1"] = pdxInstance;
            Console.WriteLine("Populated PdxInstance Object");
            IPdxInstance retPdxInstance = region["Key1"];
            if ((int)retPdxInstance.GetField("id") == p.ID
                && (int)retPdxInstance.GetField("age") == p.Age
                && (string)retPdxInstance.GetField("name") == p.Name
                && retPdxInstance.IsIdentityField("id") == true)
            {
                Console.WriteLine("PdxInstance returns all fields value expected");
            }
            else
            {
                Console.WriteLine("PdxInstance doesn't returns all fields value expected");
            }
        }
    }
}

Map .NET Domain Type Names to PDX Type Names with IPdxTypeMapper

PDX serialized instances in Java map to .NET types with the same name. If you need to adjust the .NET name, then you need to use the IPdxTypeMapper.

See the Java to .NET Type Mapping Table for current mappings.
Using IPdxTypeMapper

```csharp
// This demonstrates, how to map .NET type to pdx type or java type
public class PdxTypeMapper : IPdxTypeMapper
{
    public string ToPdxTypeName(string localTypeName)
    {
        return "pdx_" + localTypeName;
    }
    public string FromPdxTypeName(string pdxTypeName)
    {
        return pdxTypeName.Substring(4); // need to extract "pdx_"
    }
}
```
Serialize with the GemFire IGFSerializable Interface

The .NET IGFSerializable interface provides fast and compact data serialization.

**Generic and Custom Serializable Types**

All built-in generics are automatically registered at initialization. You have a couple of options for complex key types.

If your application uses more complex key types that you want to make more accessible or easier to handle, you can derive a new class from IGFSerializable. Another option is for the application to do its own object serialization using Byte[] or a custom type.

**Blobs**

If you have data that is best handled as a blob, such as structs that do not contain pointers, use a Byte[] or, if you need something more complex than Byte[], implement a custom type using either IPdxSerializable or IGFSerializable.

**Object Graphs**

If you have a graph of objects in which each node can be serializable, the parent node calls DataOutput.WriteObject to delegate the serialization responsibility to its child nodes. Similarly, your application calls DataInput.ReadObject to deserialize the object graph.

*Note:* The GemFire IGFSerializable interface does not support object graphs with multiple references to the same object. If your application uses these types of circular graphs, you must address this design concern explicitly.

For more information, see the online API documentation for DataOutput and DataInput.

**How Serialization Works with IGFSerializable**

When your application puts an object into the cache for distribution, Pivotal GemFire serializes the data by taking these steps.

1. Calls the appropriate ClassId function and creates the TypeId from it.
2. Writes the TypeId for the instance.
3. Invokes the ToData function for the instance.

When your application subsequently receives a byte array, GemFire take the following steps:

1. Decodes the TypeId and creates an object of the designated type, using the registered factory functions.
2. Invokes the FromData function with input from the data stream.
3. Decodes the data and then populates the data fields.

The TypeId is an integer of four bytes, which is a combination of ClassId integer and 0x27, which is an indicator of user-defined type.

**Implement the IGFSerializable Interface**

To store your own data types in the cache, you implement the GemFire IGFSerializable interface.

Examples follow the procedure.

**Procedure**
1. Implement the `ToData` function that serializes your data:

```csharp
void ToData(DataOutput output)
```

The `ToData` function is responsible for copying all of the data fields for the object to the object stream. The `DataOutput` class represents the output stream and provides methods for writing the primitives in a network byte order. For more about this, see the online API documentation for `DataOutput`.

2. Implement the `FromData` function that consumes a data input stream and repopulates the data fields for the object:

```csharp
void fromData (DataInput& input)
```

The `DataInput` class represents the input stream and provides methods for reading input elements. The `FromData` function must read the elements of the input stream in the same order that they were written by `ToData`. For more about this, see the online API documentation for `DataInput`.

3. Implement the `ClassId` function to return an integer which is unique for your class (in the set of all of your user-defined classes).

**Simple BankAccount Class**

This example shows a simple class, `BankAccount`, that encapsulates two `int`s:

```csharp
public class BankAccount
{
    private int m_customerId;
    private int m_accountId;
    public int Customer
    {
        get
        {
            return m_customerId;
        }
    }
    public int Account
    {
        get
        {
            return m_accountId;
        }
    }
    public BankAccount(int customer, int account)
    {
        m_customerId = customer;
        m_accountId = account;
    }
}
```

**Implementing a Serializable Class**

To make `BankAccount` serializable, you implement the `IGFSerializable` interface as shown in this example:

```csharp
public class BankAccount : IGFSerializable
{
    private int m_customerId;
    private int m_accountId;
    public int Customer
    {
        get
        {
            return m_customerId;
        }
    }
}
Register the Type

To use the `BankAccount` type, you must register it with the type system. Then, when an incoming stream contains a `BankAccount`, it can be manufactured from the associated `TypeFactoryMethod`.

```csharp
Serializable.RegisterType(BankAccount.CreateInstance);
```

Typically, you would register the type before creating the system.

Using ClassId

A `ClassId` is an integer that returns the `ClassId` of the instance being serialized. The `ClassId` is used by deserialization to determine what instance type to create and deserialize into.

Using DSFID

A `DSFID` is an integer that returns the data serialization fixed ID type. `DSFID` is used to determine what instance type to create and deserialize into. `DSFID` should not be overridden by custom implementations, and it is reserved only for built-in serializable types.
Using Custom Key Types

If your application uses its own key types that are too complex to easily force into string, you can probably improve performance by using a custom type and implementing `HashCode` and `Equals` functions. For example, if you have hybrid data types such as floating point numbers, you can implement your own type that encapsulates the floating point number. Comparing floating point numbers in this way provides greater performance than comparing a string representation of the floating point numbers, with such noticeable improvements as faster cache access and smaller payloads.

See *Serialization in Native Client Mode with a Java Server* for information about implementing key types for a native client that is used with a Java cache server.

To extend a type that implements `IPdxSerializable` or `IGFSerializable` for your key, override and implement the `HashCode` and `Equals` methods in the key as needed.
Using a Custom Class With IGFSerializable

An example shows how to use the BankAccount custom key type and the AccountHistory value type that were previously defined.

Using a BankAccount Object

class AccountHistory : IGFSerializable
{
    #region Private members
    private List<string> m_history;
    #endregion
    public AccountHistory()
    {
        m_history = new List<string>();
    }
    public void ShowAccountHistory()
    {
        Console.WriteLine("AccountHistory:");
        foreach (string hist in m_history) {
            Console.WriteLine("\t{0}", hist);
        }
    }
    public void AddLog(string entry)
    {
        m_history.Add(entry);
    }
    public static IGFSerializable CreateInstance()
    {
        return new AccountHistory();
    }
    #region IGFSerializable Members
    public IGFSerializable FromData(DataInput input)
    {
        int len = input.ReadInt32();
        m_history.Clear();
        for (int i = 0; i < len; i++) {
            m_history.Add(input.ReadUTF());
        }
        return this;
    }
    public void ToData(DataOutput output)
    {
        output.WriteInt32(m_history.Count);
        foreach (string hist in m_history) {
            output.WriteUTF(hist);
        }
    }
    public UInt32 ClassId
    {
        get
        {
            return 0x05;
        }
    }
    public UInt32 ObjectSize
    {
        get
        {
            UInt32 objectSize = 0;
            foreach (string hist in m_history) {
                objectSize += (UInt32)(hist == null ? 0 : sizeof(char) * hist.Length);
            }
            return objectSize;
        }
    }
}
public class TestBankAccount
{
    public static void Main()
    {
        // Register the user-defined serializable type.
        Serializable.RegisterType(AccountHistory.CreateInstance);
        Serializable.RegisterType(BankAccountKey.CreateInstance);
        // Create a cache.
        CacheFactory cacheFactory =
            CacheFactory.CreateCacheFactory(null);
        Cache cache = cacheFactory.Create();
        // Create a region.
        RegionFactory regionFactory =
            cache.CreateRegionFactory(RegionShortcut.CACHING_PROXY);
        Region region = regionFactory.Create("BankAccounts");
        // Place some instances of BankAccount cache region.
        BankAccountKey baKey = new BankAccountKey(2309, 123091);
        AccountHistory ahVal = new AccountHistory();
        ahVal.AddLog("Created account");
        region.Put(baKey, ahVal);
        Console.WriteLine("Put an AccountHistory in cache keyed with BankAccount.");
        // Display the BankAccount information.
        Console.WriteLine(baKey.ToString());
        // Call custom behavior on instance of AccountHistory.
        ahVal.ShowAccountHistory();
        // Get a value out of the region.
        AccountHistory history = region.Get(baKey) as AccountHistory;
        if (history != null)
        {
            Console.WriteLine("Found AccountHistory in the cache.");
            history.ShowAccountHistory();
            history.AddLog("debit $1,000,000.");
            region.Put(baKey, history);
            Console.WriteLine("Updated AccountHistory in the cache.");
        }
        // Look up the history again.
        history = region.Get(baKey) as AccountHistory;
        if (history != null)
        {
            Console.WriteLine("Found AccountHistory in the cache.");
            history.ShowAccountHistory();
        }
        // Close the cache.
        cache.Close();
    }
}

//Example 5.12 Using ICacheLoader to Load New Integers in the Region
class ExampleLoaderCallback : ICacheLoader
{
    #region Private members
    private int m_loads = 0;
    #endregion
    #region Public accessors
    public int Loads
    {
        get
        {
            return m_loads;
        }
    }
    #endregion
For region-level events, an application can use `AttributesFactory.SetCache*` methods to implement and register the `ICacheLoader`, `ICacheWriter`, and `ICacheListener` interfaces to perform custom actions.

You can use `Region.Put` for simple caching situations. For more complex needs, you should implement the `ICacheLoader` interface and allow the cache to manage the creation and loading of objects. When a `Region.Get` is called for a region entry with a value of null, the `ICacheLoader::Load` method of the cache loader (if any) for the region is invoked. A static `CacheLoader::NetSearch` method is provided which can be used by `ICacheLoader` implementations to locate the requested key in the distributed system. The `ICacheListener` interface can be used to listen to various region events after events such as create, update, or invalidate of region entries have occurred. The `ICacheWriter` interface is invoked before the events have occurred.

### Using ICacheLoader to Load New Integers in the Region

This example demonstrates an `ICacheLoader` implementation for loading new integers into a region.

```csharp
class SimpleCacheLoader<TKey, TVal> : ICacheLoader<TKey, TVal>
{
    #region ICacheLoader Members
    public TVal Load(IRegion<TKey, TVal> region, TKey key, object helper)
    {
        Console.WriteLine("SimpleCacheLoader: Received Load event for region: {0} and key: {1}", region.Name, key);
        return default(TVal);
    }
    public void Close(IRegion<TKey, TVal> region)
    {
        Console.WriteLine("SimpleCacheLoader: Received Close event of region: {0}", region.Name);
    }
    #endregion
}
```

### Using ICacheWriter to Track Creates and Updates for a Region

This example implements `ICacheWriter` to track region entry create and update events. This example just reports the events to the screen, but you can do whatever you need to do with the events.

```csharp
class SimpleCacheWriter<TKey, TVal> : ICacheWriter<TKey, TVal>
{
    #region ICacheWriter<TKey, TVal> Members
    public bool BeforeUpdate(EntryEvent<TKey, TVal> ev)
    {
        Console.WriteLine("SimpleCacheWriter: Received BeforeUpdate event for: {0}", ev.Key);
        return true;
    }
    // ... handle other entry events as needed
    public bool BeforeRegionClear(RegionEvent<TKey, TVal> ev)
    {
```
A Simple C# Example

Pivotal GemFire Native Client

Console.WriteLine("SimpleCacheWriter: Received
BeforeRegionClear event of region: {0}",
ev.Region.Name);
return true;
}
// ... handle other region events as needed
#endregion
}

A Sample ICacheListener Implementation
This example implements ICacheListener.
class SimpleCacheListener<TKey, TVal> : ICacheListener<TKey, TVal>
{
#region ICacheListener<TKey, TVal> Members
public void AfterCreate(EntryEvent<TKey, TVal> ev)
{
Console.WriteLine("SimpleCacheListener: Received AfterCreate
event
for: {0}", ev.Key);
}
// ... handle other entry events as needed
public void AfterRegionDestroy(RegionEvent<TKey, TVal> ev)
{
Console.WriteLine("SimpleCacheListener: Received
AfterRegionDestroy
event of region: {0}", ev.Region.Name);
}
// ... handle other region events as needed
#endregion
}

A Simple C# Example
An example shows how to connect to GemFire, create a cache and region, put and get keys and values,
and disconnect.
Simple C# Code
class BasicOperations
{
public static void Main(string[] args)
{
try
{
// 1. Create a cache
CacheFactory cacheFactory = CacheFactory.CreateCacheFactory();
Cache cache = cacheFactory.Create();
// 2. Create default region attributes using region factory
RegionFactory regionFactory =
cache.CreateRegionFactory(RegionShortcut.CACHING_PROXY);
// 3. Create a region
IRegion<int, string> region =
regionFactory.Create<int, string>("exampleRegion");
// 4. Put some entries
region[111] = "Value1";
region[123] = "123";

164


The .NET Framework does not find managed DLLs using the conventional PATH environment variable. In order for your assembly to find and load a managed DLL, it must either be loaded as a private assembly using assemblyBinding, or it must be installed into the Global Assembly Cache (GAC).

The GAC utility must be run on every machine that runs the .NET code.

If an assembly attempts to load the GemStone.GemFire.Cache.dll without meeting this requirement, you receive this System.IO.FileNotFoundException:

```csharp
Unhandled Exception: System.IO.FileNotFoundException: Could not load file or assembly 'GemStone.GemFire.Cache, Version=8.0.0.0, Culture=neutral, PublicKeyToken= 126e6338d9f55e0c' or one of its dependencies. The system cannot find the file specified.
File name: 'GemStone.GemFire.Cache, Version=8.0.0.0, Culture=neutral, PublicKeyT oken=126e6338d9f55e0c'
at HierarchicalClient.Main()
```
Resolving the Error

Each computer where the common language runtime is installed has a machine-wide code cache called the Global Assembly Cache (GAC). The global assembly cache stores assemblies specifically designated to be shared by several applications on the computer.

As a general guideline, keep assembly dependencies private, and locate assemblies in the application directory unless sharing an assembly is explicitly required. Share assemblies by installing them into the global assembly cache only when necessary.

To access GemStone.GemFire.Cache.dll as a private assembly, you need to specify a .config file for your application.

The file needs to be the same name as your application, with a .config suffix. For example, the .config file for main.exe would be main.exe.config. The two files must reside in the same directory.

Follow these steps to create a .config file:

1. Copy %GFCPP%/docs/default.exe.config to the appropriate location.
2. Rename default.exe.config to the name of your application.

A Sample .config File

The following example shows an excerpt of a .config file. The PublicKeyToken value is only an example, and the codebase version value is not set correctly. See %GFCPP%/docs/default.exe.config for an actual example for this release.

```xml
<configuration>
  <runtime>
    <assemblyBinding
      xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="GemStone.GemFire.Cache"
          publicKeyToken="126e6338d9f55e0c"
          culture="neutral" />
        <codeBase version="0.0.0.0"
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>

Note: If the .config file contain errors, no warning or error messages are issued. The application runs as if no .config file is present.
Implementing the Shared Assembly

Follow these steps to install the shared assembly into the Global Assembly Cache (GAC).

1. Go to the \NativeClient_\xxxx directory.
   
   cd %GFCPP%

2. Run the GAC utility to install GemStone.GemFire.Cache.dll into the GAC.
   

When you are ready to uninstall, use the /u switch. More information on the GAC utility can be found at http://www.msdn.com, or by using gacutil.exe /?.

Preserving Data

A server may preserve the data queued and intended to be sent to a native client, such that the data is not discarded if communication between the server and native client is disrupted. Preservation prevents message loss, which can cause a native client to have inconsistent data. Redundant queues and a high availability server implementation may further ensure that queued data is not lost.

There is a tradeoff between the quantity of data that a server must queue and the amount of time that the server maintains and continues to queue data intended for a native client that is not communicating with the distributed system. Client configuration specifies the amount of time that the server is to continue queueing messages. High availability permits a secondary server to assume the role of a primary server with respect to queued data in the event that the primary server no longer functions. Designation of primary and secondary servers, as well as the number of redundant copies of the queue are configurable.

High Availability for Client-Server Communication

The GemFire native client provides reliable event messaging from cache server to client to prevent data loss during server failover operations. High availability is implemented in the cache server and is configured in the native client.


**Note:** High availability functions the same whether the region is partitioned or not. See Partitioned Regions in the Pivotal GemFire User’s Guide for information about partitioned regions.
Configuring Native Clients for High Availability

Configure high availability by setting the pool attribute `subscription-redundancy` to the number of copies you want maintained.

A client maintains its queue redundancy level at the time of a primary server failure by connecting to additional secondary servers.

Native clients can specify the number of secondary servers where the client registers interest and maintains subscription channels, in addition to the subscription channel with the primary server. The secondary servers maintain redundant update queues for the client. If the primary server fails, a secondary becomes a primary to provide uninterrupted messaging to the client. If possible, another secondary is then initialized so the total number of secondaries is not reduced by the failover.

Setting the Server Redundancy Level in cache.xml

This example sets one redundant server as failover backup to the primary server:

```
<cache>
  <pool name="examplePool" subscription-enabled="true" subscription-redundancy="1">
    <server host="java_servername1" port="java_port1" />
    <server host="java_servername2" port="java_port2" />
  </pool>
  <region name = "ThinClientRegion1" >
    <region-attributes refid="CACHING_PROXY" pool-name="examplePool"/>
  </region>
</cache>
```

Setting the Server Redundancy Level Programmatically

You can set the redundancy level programmatically. This example creates a client cache with two redundant cache servers configured in addition to the primary server.

The server redundancy level can be configured using the pool API. For more information about the pool API, see Using Connection Pools.

```
PropertiesPtr pp = Properties::create( );
systemPtr = CacheFactory::createCacheFactory(pp);
// Create a cache.
cachePtr = systemPtr->setSubscriptionEnabled(true)
  ->addServer("localhost", 24680)
  ->addServer("localhost", 24681)
  ->addServer("localhost", 24682)
  ->setSubscriptionRedundancy(2)
  ->create();
```

When failover to a secondary server occurs, a new secondary is added to the redundancy set. If no new secondary server is found, the redundancy level is not satisfied but the failover procedure completes successfully. Any new live server is added as a secondary and interest is registered on it.
Sending Periodic Acknowledgment

Servers use periodic acknowledgment to reduce the likelihood of duplicate notifications, and to reduce resource usage.

When redundancy is enabled for high availability and redundancy-level is set to 1 or higher, clients send (and servers expect) periodic acknowledgment messages at configurable intervals for notifications they have received. A periodic ack is not sent by the client if there are no unacknowledged notifications at the time.

Use the following system properties in the gfcpp.properties file to configure periodic ack.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>notify-ack-interval</td>
<td>Minimum period between two consecutive acknowledgment messages sent from the client to the server. The default setting (in seconds) is 10.</td>
</tr>
<tr>
<td>notify-dupcheck-life</td>
<td>Minimum time a client continues to track a notification source for duplicates when no new notifications arrive before expiring it. The default setting (in seconds) is 300.</td>
</tr>
</tbody>
</table>

The Pool API also provides attributes to configure periodic ack and duplicate message tracking timeout. See subscription-message-tracking-timeout and subscription-ack-interval in the list of pool attributes under Configuring Pools for Servers or Locators.

Enabling Queue Conflation to Improve Update Performance

Conflation of entry update messages can reduce the number of update messages a native client receives, thereby increasing performance. The native client receives only the most recent update for a particular entry key.

Conflation is enabled for a cache server region, so all clients receiving updates for a particular region benefit from the conflation. To enable conflation, set the cache server’s enable-subscription-conflation region attribute to true. This region attribute is false by default.

The queue management code conflates entry updates as part of the enqueue operation. If the previous enqueued item for that key is also an update operation, the queue management code removes that previously enqueued update, leaving only the latest update to be sent when event distribution occurs. For high availability, conflation also occurs for any secondary queues.

Only entry update messages in a cache server region with distributed-no-ack scope are conflated. Region operations and entry operations other than updates are not conflated.

For more information, see Conflate the Server Subscription Queue in the Pivotal GemFire User’s Guide.

Overriding Queue Conflation Per-Client

Override conflation on a per-client basis by setting the conflate-events property in the native client’s gfcpp.properties file.

Valid settings are:

- server. Uses the server settings.
• true. Conflates everything sent to the client.
• false. Does not conflate anything sent to the client.

Durable Client Messaging

You can configure the redundancy level for client queues that are stored on cache servers. This ensures that the client will not lose messages if it loses the connection to its primary server.

Durable messaging allows a disconnected client application to recover its subscribed data when it reconnects to the cache server because the server continues to queue messages for which the client has registered interest.
Durable Client Messaging Requirements

The messaging queues used for durable messaging are the same regular messaging queues used for basic server-to-client messaging, with additional requirements.

Requirements, options, and functionality of messaging queues are described in the *GemFire User’s Guide*. If you are using highly available servers, see *High Availability for Client-Server Communication* for additional requirements.

For durable client messaging, you also need the following:

- **Durable clients.** If the client is durable, the server continues to maintain the client queues when the client disconnects.

  **Note:** Redundancy management is handled by the client, so when the client is disconnected from the server the redundancy of client events is not maintained. Even if the servers fail one at a time, so that running clients have time to fail over and pick new secondary servers, an offline durable client cannot fail over. As a result, the client loses its queued messages.

- **Durable interest registration.** A durable client’s interest registrations specify whether its interest in a key is durable. If it is, the servers continue queuing messages for that key while the client is disconnected.

- **Reconnection conditions.** You can program the durable client to detect whether the previously registered subscription queue is available upon reconnection and determine an approximate count of pending events in the queue. Based on the results, you can then decide whether to receive the remaining events (*Cache.readyForEvents*) or close the cache if the number is too large.

- **Cache ready message.** When it is ready to receive the stored messages, a durable client must call *Cache.readyForEvents* to send a cache ready message to the server.

- **Disconnect keepalive specification.** When a durable client disconnects normally it must tell the server whether to maintain the message queue or delete it.

- **Durable client callback method.** If you use cache listeners on the durable clients, you have the option to implement the *afterRegionLive* callback method. This callback is invoked after the durable client connects to its servers, when it has received all of its stored messages and replayed the events.
Client-Side Configuration

All durable messaging configurations are performed on the client.

**Configuring a Durable Native Client**

The durable native client can be configured in the `gfcpp.properties` file, or in the `CacheFactory::set(name, value)` call.

- **Durable client ID**—You indicate that the client is durable by giving it a `durable-client-ID`. The servers use this ID to identify the client. For a non-durable client, the `durable-client-ID` is an empty string. The ID can be any number that is unique among the clients attached to servers in the same distributed system.

- **Durable timeout**—The `durable-timeout` setting specifies how long this client’s servers should wait after the client disconnects before terminating its message queue. During that time, the servers consider the client alive and continue to accumulate messages for it. The default is 300 seconds.

  The `durable-timeout` setting is a tuning parameter. When setting the timeout, take into account the normal activity of your application, the average size of your messages, and the level of risk you can handle. Assuming that no messages are being removed from the queue, how long can the application run before the queue reaches the maximum message count? In addition, how long can it run before the queued messages consume all the memory on the client host? How serious is each of those failures to your operation?

To assist with tuning, GemFire provides statistics that track message queues for durable clients through the disconnect and reconnect cycles. For statistics documentation, see *Statistics* in the GemFire User’s Guide.

When the queue is full, it blocks further operations that add messages until the queue size drops to an acceptable level. The action to take is specified on the server. For details on configuring the queue, see *Implementing Durable Client/Server Messaging* in the GemFire User's Guide.

<table>
<thead>
<tr>
<th>Configuring a Durable Native Client Using <code>gfcpp.properties</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>The following example shows <code>gfcpp.properties</code> settings to make the client durable and set the durable timeout to 200 seconds.</td>
</tr>
<tr>
<td>```</td>
</tr>
<tr>
<td>durable-client-id=31</td>
</tr>
<tr>
<td>durable-timeout=200</td>
</tr>
<tr>
<td>```</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuring a Durable Client Through the API (C++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>This programmatic example creates a durable client using the <code>CacheFactory::set(name, value)</code> method.</td>
</tr>
<tr>
<td>```</td>
</tr>
<tr>
<td>// Create durable client's properties using the C++ api</td>
</tr>
<tr>
<td>PropertiesPtr pp = Properties::create();</td>
</tr>
<tr>
<td>pp-&gt;insert(&quot;durable-client-id&quot;, &quot;DurableClientId&quot;);</td>
</tr>
<tr>
<td>pp-&gt;insert(&quot;durable-timeout&quot;, 200);</td>
</tr>
<tr>
<td>cacheFactoryPtr = CacheFactory::createCacheFactory(pp);</td>
</tr>
<tr>
<td>```</td>
</tr>
</tbody>
</table>
**Configuring Durable Interest in Keys**

When a durable client disconnects, its servers keep queuing messages for selected keys. The client indicates which keys by registering durable interest for those keys.

This fine-grained control handles the constraints of queue size and memory by saving only the critical messages.

You still register interest for other keys, but not durable interest. When the client is connected to its servers, it receives messages for those non-durable keys. When the client is disconnected, its non-durable interest registrations are deleted but messages that are already in the queue remain there.

For durable clients, all interest registration is done immediately after the regions are created. This is required whether interest registration is durable or not durable. An extra `registerInterest` parameter specified for durable clients indicates whether the registration is durable (`true`) or not (`false`).

---

**API Client Durable Interest List Registration (C++)**

The following programmatic example registers durable interest in Key-1. The interest registration happens immediately after region creation and before anything else.

```c++
// Durable client interest registration can be
// durable (true) or nondurable (default).
VectorOfCacheableKey keys;
keys.push_back( CacheableString::create("Key-1") );
regionPtr->registerKeys(keys,true);
```

You use the typical methods for interest registration and configure notification by subscription on the server as usual. For details, see [Registering Interest for Entries](#).

**Note:** Changing interest registration after the durable client connects the first time can cause data inconsistency and is not recommended.

At restart, if the client doesn't register durable interest for exactly the same keys as before then the entries in the interest list are not copied from the server during the registration. Instead, the client cache starts out empty and entries are added during updates. If no updates come in for an entry, it never shows up in the client cache.

---

**Configuring Durable Client Reconnection**

You can configure the durable native client to obtain an approximate count of pending events upon durable client reconnection. Based on the returned number, you can determine whether to proceed and receive the pending events or whether to close the cache.

Use the `getPendingEventCount` (C++ API) and the `PendingEventCount` (C# .NET API) property to detect whether the previously registered subscription queue is available upon durable client reconnection and the count of pending events in the queue. Based on the returned results, you can then decide whether to receive the remaining events or close the cache if the number is too large.

For example, for a client with only the default pool created:

```c++
Pool pool = PoolManager.Find("PoolName");
int pendingEvents = pool.PendingEventCount;
if (pendingEvents == -2) { // client connected for the first time
    // // continue
} else if (pendingEvents == -1) { // client reconnected but after the timeout period
    // handle possible data loss
} else { // pendingEvents >= 0
    // // decide to invoke readyForEvents() or cache.close(false)/pool.destroy()
}
```
For a client with multiple pools:

```java
int pendingEvents = 0;
int pendingEvents1 = PoolManager.Find("pool1").PendingEventCount;
pendingEvents += (pendingEvents1 > 0) ? pendingEvents1 : 0;
int pendingEvents2 = PoolManager.Find("pool2").PendingEventCount;
pendingEvents += (pendingEvents2 > 0) ? pendingEvents2 : 0;
// process individual pool counts separately.
```

The `getPendingEventCount` method and `PendingEventCount` property can return the following possible values:

- A value representing a count of events pending at the server. Note that this count is an approximate value based on the time the durable client pool connected or reconnected to the server. Any number of invocations will return the same value.
- A zero value if there are no events pending at server for this client pool.
- A negative value indicates that no queue is available at the server for the client pool.
  - -1 indicates that the client pool has reconnected to the server after its durable-client-timeout period has elapsed. The pool's subscription queue has been removed possibly causing data loss.
  - A value of -2 indicates that this client pool has connected to server for the first time.
Sending Cache Ready Messages to the Server

After a durable client connects and initializes its cache, regions, cache listeners, and any interest registration, it invokes `readyForEvents` to indicate to the servers that the client is ready to receive any messages accumulated for it.

### Durable Client Cache Ready Notification (C++)

The following example shows how to call `readyForEvents`.

```cpp
//Send ready for event message to server (only for durable clients).
//Server will send queued events to client after receiving this.
cachePtr->readyForEvents();
```

To keep the client from losing events, do not call this method until all regions and listeners are created. For more information, see `Reconnection`. 
Disconnecting from the Server

When a durable client closes its cache and disconnects, it tells the servers whether to maintain its queues.

For this purpose, use the version of `Cache::close` with the boolean `keepalive` parameter set, as shown in the following example. If the setting is true, the servers keep the durable client’s queues and durable subscriptions alive for the timeout period. In addition to in-memory queue retention, the servers can evict the most recent durable client queue updates to disk to reduce memory consumption.

Only the resources and data related to the session are removed, such as port numbers and non-durable subscriptions. If the setting is false, the servers do the same cleanup that they would do for a nondurable client.

```
Durable Client Disconnect With Queues Maintained

// Close the Cache and disconnect with keepalive=true.
// Server will queue events for durable registrations and CQs
// When the client reconnects (within a timeout period) and sends
// "readyForEvents()", the server will deliver all stored events
cachePtr->close(true);
```
Life Cycle of a Durable Client

This section discusses the high-level operation of a durable client through initial startup, disconnection, and reconnection.

Initial Operation

The initial startup of a durable client is similar to the startup of any other client, except that it specifically calls the `Cache.readyForEvents` method when all regions and listeners on the client are ready to process messages from the server.

See Sending the Cache Ready Message to the Server.

Disconnection

While the client and servers are disconnected, their operation varies depending on the circumstances.

Normal disconnect

When a durable client disconnects normally, the `Cache.close` request states whether to maintain the client's message queue and durable subscriptions. The servers stop sending messages to the client and release its connection. See Disconnecting From the Server for more information.

If requested, the servers maintain the queues and durable interest list until the client reconnects or times out. The non-durable interest list is discarded. The servers continue to queue up incoming messages for entries on the durable interest list. All messages that were in the queue when the client disconnected remain in the queue, including messages for entries on the non-durable list.

If the client requests to not have its subscriptions maintained, or if there are no durable subscriptions, the servers unregister the client and perform the same cleanup as for a non-durable client.

Abnormal disconnect

If the client crashes or loses its connections to all servers, the servers automatically maintain its message queue and durable subscriptions until the client reconnects or times out.

Client disconnected but operational

If the client operates while it is disconnected, it gets what data it can from the local cache. Since updates are not allowed, the data can become stale. An `UnconnectedException` occurs if an update is attempted.

Timing out while disconnected

The servers track how long to keep a durable client queue alive based on the `durable-client-timeout` setting. If the client remains disconnected longer than the timeout, the servers unregister the client and do the same cleanup that is performed for a non-durable client. The servers also log an alert. When a timed-out client reconnects, the servers treat it as a new client making its initial connection.

Reconnection

During initialization, operations on the client cache can come from multiple sources.

- Cache operations by the application.
- Results returned by the cache server in response to the client's interest registrations.
- Callbacks triggered by replaying old events from the queue.

These procedures can act on the cache concurrently, and the cache is never blocked from doing operations.
GemFire handles the conflicts between the application and interest registration, but you need to prevent the callback problem. Writing callback methods that do cache operations is never recommended, but it is a particularly bad idea for durable clients, as explained in Implementing Cache Listeners for Durable Clients on page 161.

Program the durable client to perform these steps, in order, when it reconnects:

1. Create the cache and regions. This ensures that all cache listeners are ready. At this point, the application hosting the client can begin cache operations.
2. Issue its register interest requests. This allows the client cache to be populated with the initial interest registration results. The primary server responds with the current state of those entries if they still exist in the server's cache.
3. Call Cache.readyForEvents. This tells the servers that all regions and listeners on the client are now ready to process messages from the servers. The cache ready message triggers the queued message replay process on the primary server.

For an example that demonstrates Cache.readyForEvents, see Sending the Cache Ready Message to the Server.

This figure shows the concurrent procedures that occur during the initialization process. The application begins operations immediately on the client (step 1), while the client's cache ready message (also step 1) triggers a series of queue operations on the cache servers (starting with step 2 on the primary server). At the same time, the client registers interest (step 2 on the client) and receives a response from the server.

Message B2 applies to an entry in Region A, so the cache listener handles B2's event. Because B2 comes before the marker, the client does not apply the update to the cache.
Preserving Data

Figure 6: Initialization of a Reconnected Durable Client

Only one region is shown for simplicity, but the messages in the queue could apply to multiple regions. Also, the figure omits the concurrent cache updates on the servers, which would normally be adding more messages to the client's message queue.

Durable Message Replay

When the primary server receives the cache ready message, the servers and client execute a procedure to update the queue and replay the events from the stored messages.

Durable message replay proceeds as follows. To avoid overwriting current entries with old data, the client does not apply the updates to its cache.

1. The server finds the queue for this durable client ID and updates its information, including the client's socket and remote ports.

   If the client has timed out while it was disconnected, its queues are gone and the server then treats it as a new client. See Initial Operation on page 157.

2. All servers that have a queue for this client place a marker in the queue.

   Messages in the queue before the marker are considered to have come while the client was disconnected. Messages after the marker are handled normally.
3. The cache server sends the queued messages to the client. This includes any messages that were evicted to disk.

4. The client receives the messages but does not apply the updates to its cache. If cache listeners are installed, they handle the events. For implications, see Implementing Cache Listeners for Durable Clients.

5. The client receives the marker message indicating that all past events have been played back.

6. The cache server sends the current list of live regions.

7. In each live region on the client, the marker event triggers the afterRegionLive callback.

   After the callback, the client begins normal processing of events from the server and applies the updates to its cache.

Even when a new client starts up for the first time, the cache ready markers are inserted in the queues. If messages start coming into the new queues before the servers insert the marker, those messages are considered as having happened while the client was disconnected, and their events are replayed the same as in the reconnect case.

**Application Operations During Interest Registration**

As soon as the client creates its regions, the application hosting the client can start cache operations, even while the client is still receiving its interest registration responses.

In that case, application operations take precedence over interest registration responses.

When adding register interest responses to the cache, the following rules are applied:

- If the entry already exists in the cache with a valid value, it is not updated.
- If the entry is invalid and the register interest response is valid, the valid value is put into the cache.
- If an entry is marked destroyed, it is not updated. Destroyed entries are removed from the system after the register interest response is completed.

If the interest response does not contain any results because all of those keys are absent from the server’s cache, the client’s cache can start out empty. If the queue contains old messages related to those keys, the events are still replayed in the client’s cache.
Implementing Cache Listeners for Durable Clients

A cache listener for durable clients requires all callback methods to behave properly when stored events are replayed. A cache listener has a callback method, `afterRegionLive`, specifically for durable clients aspects.

For general instructions on implementing a cache listener, see `CacheListener`.

Writing Callbacks for Use With Durable Messaging

Durable clients require special attention to cache callbacks generated by the cache listener. During the initialization window when a reconnecting client has a functioning cache but is still receiving the stored messages from the queue, the client can replay events that are long past. These events are not applied to the cache, but they are sent to the cache listener. If the listener’s callbacks invoked by these events make changes to the cache, that could conflict with current operations and create data inconsistencies.

Consequently, you need to keep your callback implementations lightweight and not do anything in the cache that could produce incorrect results during this window. For details on implementing callbacks for GemFire event handlers, see `Implementing Cache Event Handlers` in the GemFire User’s Guide.

Implementing the afterRegionLive Method

If you are using cache listeners, you can implement the `afterRegionLive` callback method provided for durable clients. This callback is invoked when the client has received all the old messages that were stored in its queue while it was disconnected. Implementing this method enables you to do application-specific operations when the client has replayed all of these old events.

If you do not wish to use this callback, and your listener is an instance of `CacheListener` (not a `CacheListenerAdapter`), you must implement `afterRegionLive` as a non-operational method.
Part XI

Security

Security describes how to implement the security framework for the Pivotal GemFire native client, including authentication, authorization, encryption, and SSL client/server communication.

The security framework authenticates clients that attempt to connect to a GemFire cache server, and authorizes client cache operations. You can also configure it for client authentication of servers, and you can plug in your own implementations for authentication and authorization.

Authentication

A client is authenticated when it connects, with valid credentials, to a GemFire cache server that is configured with the client Authenticator callback.

Once the client is authenticated, the server assigns the client a unique ID and principal, used to authorize operations. The client must trust all cache servers in the server system as it may connect to any one of them. For information on configuring client/server, see Client/Server Configuration in the Topologies and Communication section of the Pivotal GemFire User's Guide.
Process and Multiuser Authentication

Client connections can be authenticated at two levels, process and multiuser.

- **Process.** Each pool creates a configured minimum number of connections across the server group. The pool accesses the least-loaded server for each cache operation.

  Process-level connections represent the overall client process and are the standard way a client accesses the server cache.

- **Multi-user.** Each user/pool pair creates a connection to one server and then sticks with it for operations. If the server is unable to respond to a request, the pool selects a new one for the user.

  Typically, application servers or web servers that act as clients to GemFire servers make multi-user connections. Multi-user allows a single application or web server process to service a large number of users with varied access permissions.

By default, server pools use process-level authentication. Enable multi-user authentication by setting a pool’s `multi-user-secure-mode-enabled` attribute to `true`.

Credentials can be sent in encrypted form using the Diffie-Hellman key exchange algorithm. See *Encrypt Credentials with Diffie-Hellman* for more information.
Configuring Credentials for Authentication

The native client uses system properties to acquire valid credentials for authentication by the server. You define these properties in the `gfcpp.properties` file, which the native client accesses during startup.

- `security-client-auth-factory`
- `security-client-auth-library`
- Implementing the Factory Method for Authentication (C++ and .NET)
- Acquiring Credentials Programmatically (C++ and .NET)

**security-client-auth-factory**

System property for the factory function of the class implementing the `AuthInitialize` interface (`IAuthInitialize` in .NET). The .NET clients can load both C++ and .NET implementations. For .NET implementations, this property is the fully qualified name of the static factory function (including the namespace and class).

**security-client-auth-library**

System property for the library where the factory methods reside. The library is loaded explicitly and the factory functions are invoked dynamically, returning an object of the class implementing the `AuthInitialize` interface.

Other implementations of the `AuthInitialize` interface may be required to build credentials using properties that are also passed as system properties. These properties also start with the security-prefix. For example, the PKCS implementation requires an alias name and the corresponding keystore path, which are specified as `security-alias` and `security-keystorepath`, respectively. Similarly, `UserPasswordAuthInit` requires a username specified in `security-username`, and the corresponding password is specified in the `security-password` system property.

The `getCredentials` function for the `AuthInitialize` interface is called to obtain the credentials. All system properties starting with `security-` are passed to this callback as the first argument to the `getCredentials` function, as shown in the following code snippet:

```cpp
PropertiesPtr getCredentials(PropertiesPtr& securityprops, const char *server);
```

**Implementing the Factory Method for Authentication (C++ and .NET)**

The following examples show how to implement the factory method in both C++ and .NET.

**C++ Implementation**

```cpp
LIBEXP AuthInitialize* createPKCSAuthInitInstance()
{
    return new PKCSAuthInit();
}
```

**.NET Implementation**

```csharp
public static IAuthInitialize Create()
{
    return new UserPasswordAuthInit();
}
```

Implementations of the factory method are user-provided. Credentials in the form of properties returned by this function are sent by the client to the server for authentication during the client’s handshake process with the server.
The GemFire native client installation provides sample security implementations in its templates/security folder.

**Acquiring Credentials Programmatically (C++ and .NET)**

This example shows a C++ client connecting with credentials.

```cpp
PropertiesPtr secProp = Properties::create();
secProp->insert("security-client-auth-factory", "createPKCSAuthInitInstance");
secProp->insert("security-client-auth-library", "securityImpl");
secProp->insert("security-keystorepath", "keystore/gemfire6.keystore");
secProp->insert("security-alias", "gemfire6");
secProp->insert("security-zkeystorepass", "gemfire");
CacheFactoryPtr cacheFactoryPtr = CacheFactory::createCacheFactory(secProp);
```

This example shows a .NET client.

```csharp
Properties secProp = Properties.Create();
secProp.Insert("security-client-auth-library", "securityImpl");
secProp.Insert("security-username", "gemfire6");
secProp.Insert("security-password", "gemfire6Pass");
```
Configuring Authentication by the Cache Server

When the cache server receives client credentials during the handshake operation, the server authenticates the client with the callback configured in the `security-client-authenticator` system property. The handshake succeeds or fails depending on the results of the authentication process.

Here is an example of how you could configure `security-client-authenticator` in the `gfcpp.properties` file:

```
security-client-authenticator=templates.security.PKCSAuthenticator.create
```

In the preceding configuration sample, `PKCSAuthenticator` is the callback class implementing the `Authenticator` interface and `create` is its factory method.

The following example shows an implementation of the static `create` method:

```java
public static Authenticator create() {
    return new PKCSAuthenticator();
}
```
Server Authentication Errors

An AuthenticationRequiredException is thrown when the server is configured with security and the client does not present its credentials while attempting to connect. This can occur if the `security-client-auth-factory` and `security-client-auth-library` properties are not configured on the client.
Creating Multiple Secure User Connections

To create multiple, secure connections to your servers from a single client, so the client can service different user types, you create an authenticated `RegionService` for each user.

Typically, a GemFire client embedded in an application server supports data requests from many users. Each user can be authorized to access a subset of data on the servers. For example, customer users are allowed only to see and update their own orders and shipments.

The authenticated users all access the same Cache through instances of the `RegionService` interface. See `RegionService`.

To implement multiple user connections in your client cache, create your Cache as usual, with these additions:

1. Configure your client’s server pool for multiple secure user authentication. Example:

   ```xml
   <pool name="serverPool" multiuser-authentication="true">
   <locator host="host1" port="44444"/>
   </pool>
   ```

   This enables access through the pool for the `RegionService` instances and disables it for the Cache instance.

2. After you create your cache, for each user, call your Cache instance `createAuthenticatedView` method, providing the user’s particular credentials. These are create method calls for two users:

   ```cpp
   PropertiesPtr credentials1 = Properties::create();
   credentials1->insert("security-username", "root1");
   credentials1->insert("security-password", "root1");
   RegionServicePtr userCache1 = cachePtr->createAuthenticatedView( credentials1 );
   
   PropertiesPtr credentials2 = Properties::create();
   credentials2->insert("security-username", "root2");
   credentials2->insert("security-password", "root2");
   RegionServicePtr userCache2 = cachePtr->createAuthenticatedView( credentials2 );
   ```

   For each user, do all of your caching and region work through the assigned region service pointer. Use the region service to get your regions, and the query service, if you need that, and then do your work with them. Access to the server cache will be governed by the server’s configured authorization rules for each individual user.

3. To close your cache, close the Cache instance.

Requirements and Caveats for `RegionService`

For each region, you can perform operations through the `Cache` instance or the `RegionService` instances, but not both.

**Note:** Through the `Cache` you can create a region that uses a pool configured for multi-user authentication, then access and do work on the region using your `RegionService` instances.

To use `RegionService`:

- Configure regions as EMPTY. Depending on your data access requirements, this configuration might affect performance, because the client goes to the server for every `get`.

- If you are running durable CQs through the region services, stop and start the offline event storage for the client as a whole. The server manages one queue for the entire client process, so you need to request the stop and start of durable client queue (CQ) event messaging for the cache as a whole,
through the ClientCache instance. If you closed the RegionService instances, event processing would stop, but the events from the server would continue, and would be lost.

Stop with:

```cpp
  cachePtr->close(true);
```

Start up again in this order:

1. Create the cache.
2. Create all region service instances. Initialize CQ listeners.
3. Call the cache `readyForEvents` method.
Using an LDAP Server for Client Authentication

An LDAP server can be used by a GemFire cache server using the sample LDAP implementation provided in GemFire server product.

See the Security chapter in the GemFire User's Guide to verify authentication credentials for native clients attempting to connect to the GemFire servers and sending username and passwords using the sample UserPassword scheme.

Note: The username and password with this sample implementation is sent out in plaintext. For better security, either turn on credential encryption using Diffie-Hellman key exchange, or use a scheme like PKCS.

When a client initiates a connection to a cache server, the client submits its credentials to the server and the server submits those credentials to the LDAP server. To be authenticated, the credentials for the client need to match one of the valid entries in the LDAP server. The credentials can consist of the entry name and the corresponding password. If the submitted credentials result in a connection to the LDAP server because the credentials match the appropriate LDAP entries, then the client is authenticated and granted a connection to the server. If the server fails to connect to the LDAP server with the supplied credentials then an AuthenticationFailedException is sent to the client and its connection with the cache server is closed.

Configuration Settings

In the gfcpp.properties file for the client, specify the UserPasswordAuthInit callback, the username, and the password, like this:

```plaintext
security-client-auth-library=securityImpl
security-client-auth-factory=createUserPasswordAuthInitInstance
security-username=<username>
security-password=<password>
```

For server side settings and LDAP server configuration, see the Security chapter in the Pivotal GemFire User's Guide.

Encrypted Authentication

You can set up encrypted authentication using Diffie-Hellman or the sample PKCS implementation.
Encrypt Credentials with Diffie-Hellman

For secure transmission of sensitive credentials like passwords, encrypt credentials using the Diffie-Hellman key exchange algorithm. With Diffie-Hellman enabled, you can have your client authenticate its servers.

- Enabling Diffie-Hellman
- Client Authentication of Server

Enabling Diffie-Hellman

Set the `security-client-dhalgo` system property in the `gfcpp.properties` file to the password for the public key file store on the client (the name of a valid symmetric key cipher supported by the JDK).

Valid `security-client-dhalgo` property values are DESede, AES, and Blowfish, which enable the Diffie-Hellman algorithm with the specified cipher to encrypt the credentials.

For the AES and Blowfish algorithms, optionally specify the key size for the `security-client-dhalgo` property. Valid key size settings for the AES algorithm are AES:128, AES:192, and AES:256. The colon separates the algorithm name and the key size. For the Blowfish algorithm, key sizes from 128 to 448 bits are supported. For example:

```
security-client-dhalgo=Blowfish:128
```

For AES algorithms, you may need Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files from Sun or equivalent for your JDK.

Adding settings for Diffie-Hellman on clients also enables challenge response from server to client in addition to encryption of credentials using the exchanged key to avoid replay attacks from clients to servers. Clients can also enable authentication of servers, with challenge-response from client to server to avoid server-side replay attacks.

Client Authentication of Server

With Diffie-Hellman enabled, you can have your client authenticate its servers.

1. Generate a `.pem` file for each pkcs12 keystore:
   a. Enter this command from a pkcs12 file or a pkcs keystore:

   ```
   user@host: ~> openssl pkcs12 -nokeys -in <keystore/pkcs12 file> -out <outputfilename.pem >
   ```
   b. Concatenate the generated `.pem` files into a single `.pem` file. You will use its name in the next step.

2. In the `gfcpp.properties` file:
   a. Set `security-client-kspath` to the name of the `.pem` file password for the public key file store on the client.
   b. Set `security-client-kspasswd` to the password for the public key file store on the client.
Using PKCS for Encrypted Authentication

This section discusses the concepts and configurations for the sample UserPassword and PKCS implementations. Descriptions of their interfaces, classes, and methods are available in the online API.

Note: Native client samples are provided in source form only in the "templates" directory within the product directory.

Disclaimer: These security samples serve only as example implementations. The implementation and its source code is provided on an "as-is" basis, without warranties or conditions of any kind, either express or implied. You can modify these samples to suit your specific requirements and security providers. GemStone Systems, Inc. takes no responsibility and accepts no liability for any damage to computer equipment, companies or personnel that might arise from the use of these samples.

With PKCS, clients send encrypted authentication credentials in the form of standard PKCS signatures to a GemFire cache server when they connect to the server. The credentials consist of the alias name and digital signature created using the private key that is retrieved from the provided keystore. The server uses a corresponding public key to decrypt the credentials. If decryption is successful then the client is authenticated and it connects to the cache server. For unsuccessful decryption, the server sends an AuthenticationFailedException to the client, and the client connection to the cache server is closed.

When clients require authentication to connect to a cache server, they use the PKCSAuthInit class implementing the AuthInitialize interface to obtain their credentials. For the PKCS sample provided by GemFire, the credentials consist of an alias and an encrypted byte array. The private key is obtained from the PKCS#12 keystore file. To accomplish this, PKCSAuthInit gets the alias retrieved from the security-alias property, and the keystore path from the security-keystorepath property. PKCSAuthInit also gets the password for the password-protected keystore file from the security-keystorepass property so the keystore can be opened.

Building the securityImpl Library

To use the PKCS sample implementation, you need to build OpenSSL and then build the securityImpl library. In the gfcpp.properties file for the client, specify the PKCSAuthInit callback, the keystore path, the security alias, and the keystore password, like this:

```
security-client-auth-library=securityImpl
security-client-auth-factory=createPKCSAuthInitInstance
security-keystorepath=<PKCS#12 keystore path>
security-alias=<alias>
security-keystorepass=<keystore password>
```

For server side settings, see the description of PKCS sample in the Security chapter in the GemFire User’s Guide.

Client Authorization

Using a provided callback that implements the AccessControl interface, you can configure each server to authorize some or all cache operations.

The callback can also modify or even disallow the data being provided by the client in the operation, such as a put or a putAll operation. The callback can also register itself as a post-processing filter that is passed operation results like get, getAll, and query.
Configuring Client Authorization

You can configure authorization on a per-client basis for various cache operations such as creates, gets, puts, query invalidations, interest registration, and region destroys. On the server side, the `security-client-accessor` system property in the server’s `gemfire.properties` file specifies the authorization callback.

For example:

```
security-client-accessor=templates.security.XmlAuthorization.create
```

In the preceding system property setting, `XmlAuthorization` is the callback class that implements the `AccessControl` interface. The `XmlAuthorization` sample implementation provided with GemFire expects an XML file that defines authorization privileges for the clients. For details of this sample implementation and the `AccessControl` interface, see the Authorization Example section in the Security chapter for the GemFire User’s Guide.
Post-Operative Authorization

Authorization in the post-operation phase occurs on the server after the operation is complete and before the results are sent to the client.

The callback can modify the results of certain operations, such as query, get and keySet, or even completely disallow the operation. For example, a post-operation callback for a query operation can filter out sensitive data or data that the client should not receive, or even completely fail the operation.

The `security-client-accessor-pp` system property in the server's `gemfire.properties` file specifies the callback to invoke in the post-operation phase. For example:

```
security-client-accessor-pp=templates.security.XmlAuthorization.create
```
Determining Pre- or Post-Operation Authorization

The `OperationContext` object that is passed to the `authorizeOperation` method of the callback as the second argument provides an `isPostOperation` method that returns true when the callback is invoked in the post-operation phase.

For example:

```cpp
bool authorizeOperation(Region region, OperationContext context) {
  if (context.isPostOperation()) {
    // It's a post-operation
  } else {
    // It's a pre-operation
  }
}
```

If an authorization failure occurs in a pre-operation or post-operation callback on the server, the operation throws a `NotAuthorizedException` on the client.

For more information, see `Authorization` in the GemFire User’s Guide.

---

Security-Related System Properties (gfcpp.properties)

The table describes the security-related system properties in the `gfcpp.properties` file for native client authentication and authorization.

**Table 19: System Properties for Client Authentication and Authorization**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>security-client-auth-factory</code></td>
<td>Sets the key for the <code>AuthInitialize</code> factory function.</td>
</tr>
<tr>
<td><code>security-client-auth-library</code></td>
<td>Registers the path to the <code>securityImpl.dll</code> library.</td>
</tr>
<tr>
<td><code>security-client-dhalgo</code></td>
<td>Returns the Diffie-Hellman secret key cipher algorithm.</td>
</tr>
<tr>
<td><code>security-client-kspath</code></td>
<td>Path to a .pem file, which contains the public certificates for all GemFire cache servers to which the client can connect through specified endpoints.</td>
</tr>
<tr>
<td><code>security-client-kspassword</code></td>
<td>Password for the public key file store on the client.</td>
</tr>
<tr>
<td><code>security-keystorepath</code></td>
<td>Path to the public keystore.</td>
</tr>
<tr>
<td><code>security-alias</code></td>
<td>Alias name for the key in the keystore.</td>
</tr>
<tr>
<td><code>security-keystorepass</code></td>
<td>Sets the password for the password-protected keystore.</td>
</tr>
<tr>
<td><code>ssl-enabled</code></td>
<td>Enables SSL-based client/server communication when set to true. When true, the other <code>ssl-*</code> settings are required. The default is false, which causes communication to use plain socket connections.</td>
</tr>
<tr>
<td><code>ssl-keystore</code></td>
<td>Name of the .PEM keystore file, containing the client’s private key. Not set by default. Required if <code>ssl-enabled</code> is true.</td>
</tr>
</tbody>
</table>
SSL Client/Server Communication

This section describes how to configure OpenSSL; implement SSL-based communication between your clients and servers; and run clients and servers with SSL enabled.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssl-keystore-password</td>
<td>Sets the password for the private key PEM file for SSL.</td>
</tr>
<tr>
<td>ssl-truststore</td>
<td>Name of the .PEM truststore file, containing the servers’ public certificate. Not set by default. Required if ssl-enabled is true.</td>
</tr>
</tbody>
</table>
Step 1: Download and install OpenSSL

The open-source OpenSSL toolkit provides a full-strength general purpose cryptography library to operate along with the PKCS sample implementation for encrypted authentication of native client credentials.

Follow these instructions to download and install OpenSSL for your specific operating system.

The GemFire native client requires OpenSSL 1.0.1s or later. For Windows platforms, you can use either the regular or the OpenSSL 1.0.1s "Light" version.

**Note:** If you use Cygwin, it is recommended that you do not use the OpenSSL library that comes with Cygwin because it is built with cygwin.dll as a dependency.

**Linux**

Download the OpenSSL tarball archive from the OpenSSL web site at [http://www.openssl.org/source/](http://www.openssl.org/source/).
Copy the downloaded tarball file into `NativeClient_xxxx/templates/security/openssl/Linux` and run `buildit.sh`.

**Solaris**

Download the OpenSSL tarball archive from the OpenSSL web site at [http://www.openssl.org/source/](http://www.openssl.org/source/).
Copy the downloaded tarball file into `NativeClient_xxxx/templates/security/openssl/SunOS` and run `buildit.sh`.

**Windows**

Download the installer for OpenSSL from [http://www.openssl.org/related/binaries.html](http://www.openssl.org/related/binaries.html). You can also use the OpenSSL "Light" version.

Use the downloaded OpenSSL installer to install it on Windows. You can usually accept the default installation path (`C:\OpenSSL`).
Step 2: Create keystores

The GemFire server requires keys and keystores in the Java Key Store (JKS) format while the native client requires them in the clear PEM format. Thus you need to be able to generate private/public keypairs in either format and convert between the two using the `keytool` utility and the `openssl` command.

There are public third party free tools and source code available to download such as the "KeyTool IU" tool.
Step 3: Configure environment variables

Configure your system environment to build and run OpenSSL. Follow the environment setup that applies to your operating system.

For all references to the Pivotal_GemFire_NativeClient_32or64bit_xxxx directory, replace 32or64bit with the appropriate architecture and xxxx with the actual four-digit product build number.

Bourne and Korn shells (sh, ksh, bash)

% OPENSSL=<parent folder for OpenSSL binaries>; export OPENSSL
% GFCPP=<path to installation, typically C:\Pivotal_GemFire_NativeClient_32or64bit_xxxx>; export GFCPP
% LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$GFCPP/lib:$GFCPP/ssl_libs:$OPENSSL/lib
% export LD_LIBRARY_PATH
% CLASSPATH=$GEMFIRE/lib/gfSecurityImpl.jar:$CLASSPATH

Windows

> set GFCPP=<path to installation, typically C:\Pivotal_GemFire_NativeClient_32or64bit_xxxx>
> set OPENSSL=<path to installed OpenSSL, typically C:\OpenSSL>
> set PATH=<path to Java JDK or JRE>\bin;%GFCPP%\bin;%GFCPP%\ssl_libs;%OPENSSL%\bin;
%PATH%
> set CLASSPATH=<path to GemFire installation>\lib\gfSecurityImpl.jar;%CLASSPATH%
Step 4: Configure SSL properties in gfcpp.properties and gemfire.properties

Configure SSL properties.

1. In gfcpp.properties, set ssl-enabled to true and set ssl-keystore and ssl-truststore to point to your keystore files. See Security-Related System Properties (gfcpp.properties) for a description of these properties.

2. On each locator, set the following SSL properties in the locator's gemfire.properties file:

```
server-ssl-enabled=true
server-ssl-protocols=any
server-ssl-require-authentication=true
server-ssl-ciphers=SSL_RSA_WITH_NULL_SHA
```
Step 5: Start and stop the client and server

Before you start and stop the client and server, make sure you configure the native client with the SSL properties as described and with the servers or locators specified as usual.

Specifically, ensure that:

- OpenSSL and ACE_SSL DLLs locations are in the right environment variables for your system: PATH for Windows, and LD_LIBRARY_PATH for Unix.
- You have generated the keys and keystores.
- You have set the system properties.

For details on stopping and starting locators and cache servers with SSL, see Starting Up and Shutting Down Your System in the GemFire User’s Guide.

Example locator start command

Ensure that all required SSL properties are configured in your server's gfsecurity.properties file. Then start your locator as follows:

```
gfsh>start locator --name=my_locator --port=12345 --dir=. \ 
--security-properties-file=/path/to/your/gfsecurity.properties
```

Example locator stop command

```
gfsh>stop locator --port=12345 \ 
--security-properties-file=/path/to/your/gfsecurity.properties
```

Example server start command

Again, ensure that all required SSL properties are configured in gfsecurity.properties. Then start the server with:

```
gfsh>start server --name=my_server --locators=hostname[12345] \ 
--cache-xml-file=server.xml --log-level=fine \ 
--security-properties-file=/path/to/your/gfsecurity.properties
```

Example server stop command

```
gfsh>stop server --name=my_server
```
Limitations

Currently the native client only supports the NULL cipher with mutual authentication for SSL socket communications.

The keys and keystores need to be in the JKS (Java KeyStore) format for the GemFire server and in the clear PEM format for the native client.
Remote Querying

Remote Querying documents remote querying from the native client to the GemFire cache server. Using examples and procedures, it describes how to use the APIs to run queries against cached data; work with query strings in the native client; create and manage queries; and create indexes.

Remote Querying Basics

Use the Pivotal GemFire native client query API to query your cached data stored on a GemFire cache server. The query is evaluated and executed on the cache server, and the results are returned to the native client.

You can also optimize your queries by defining indexes on the cache server.

The query language for the native client is essentially a subset of OQL (ODMG 3.0 Object Data Management Group, http://www.odmg.org), which is based on SQL-92. OQL is a SQL-like language with extended functionality for querying complex objects, object attributes, and methods.

It is assumed that you have general familiarity with SQL querying and indexing, and with the information on the native client cache provided in previous chapters of this guide.

The online C++ and .NET API documentation located in the docs directory for the native client provides extensive details for all of the querying interfaces, classes and methods.

Query language features and grammar are described in detail in the Querying chapter of the GemFire User's Guide. This chapter describes areas that are unique to the native client.

If you are using the new pool API, you should obtain the QueryService from the pool. For information about the pool API, see Native Client Pool API.
Examples Data and Class Definitions

Examples show C++ and corresponding Java class definitions and sample data for the /portfolios region. The region’s keys are the portfolio ID.

User-defined data types must implement the `Serializable` interface on the native client side, while corresponding Java classes must implement the `DataSerializable` interface. The C++ objects for the native client must correspond to the Java objects for the GemFire cache server. This means that an object on one side should deserialize correctly to the other side.

**Sample C++ class definition**

```cpp
class Portfolio : public Serializable {
    int ID;
    char * type;
    char * status;
    Map<Position> positions;
}
class Position : public Serializable {
    char * secId;
    double mktValue;
    double qty;
}
```

**Corresponding Java class definition**

```java
class Portfolio implements DataSerializable {
    int ID;
    String type;
    String status;
    Map positions;
}
class Position implements DataSerializable {
    String secId;
    double mktValue;
    double qty;
}
```

The following table lists the sample data in the /portfolios region.

**Table 20: Entry values in /portfolios**

<table>
<thead>
<tr>
<th>id</th>
<th>type</th>
<th>Statusted</th>
<th>Position: secID</th>
<th>Position: mktValue</th>
<th>Position: qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>xyz</td>
<td>active</td>
<td>xxx</td>
<td>27.34</td>
<td>1000.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>xxy</td>
<td>26.31</td>
<td>1200.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>xxz</td>
<td>24.30</td>
<td>1500.00</td>
</tr>
<tr>
<td>222</td>
<td>xyz</td>
<td>active</td>
<td>yyy</td>
<td>18.29</td>
<td>5000.00</td>
</tr>
<tr>
<td>333</td>
<td>abc</td>
<td>active</td>
<td>aaa</td>
<td>24.30</td>
<td>10.00</td>
</tr>
<tr>
<td>333</td>
<td>abc</td>
<td>active</td>
<td>aab</td>
<td>23.10</td>
<td>15.00</td>
</tr>
<tr>
<td>444</td>
<td>abc</td>
<td>inactive</td>
<td>bbb</td>
<td>50.41</td>
<td>100.00</td>
</tr>
<tr>
<td>444</td>
<td>abc</td>
<td>inactive</td>
<td>bbc</td>
<td>55.00</td>
<td>90.00</td>
</tr>
</tbody>
</table>
Because the client cache waits during transaction execution, and client regions are not distributed, the only activities that interact with a client transaction are those that occur on the server.
Executing a Query from the Native Client

**C#/.NET and C++ examples show how to execute a query from the native client.**

**Note:** In all queries that use the example data, it is assumed that the `/portfolios` region has `javaobject.Portfolio` objects on the cache server.

1. If you are using the C++ native client, get a pointer to the `QueryService` method.
2. Create a `QueryPtr` to a query (C++) or create a query instance (C# .NET) that is compatible with the OQL specification.
3. Use the `execute` method for the `Query` interface to submit the query string to the cache server. The server remotely evaluates the query string and returns the results to the client.
4. You can iterate through the returned objects as part of the query process.

**C#/.NET Example**

```
Query<Portfolio> qry = qrySvc.NewQuery("SELECT DISTINCT * FROM /Portfolios");
ISelectResults<Portfolio> results = qry.Execute();
SelectResultsIterator<Portfolio> iter = results.GetIterator();
while (iter.MoveNext()) {
    Console.WriteLine( iter.Current.ToString()); }
```

**C++ Example**

**Note:** The C++ examples in this chapter all assume that you have already obtained a pointer to the `QueryService`.

```
QueryServicePtr qrySvcPtr = cachePtr->getQueryService("examplePool");
QueryPtr qry = qrySvcPtr->newQuery("SELECT DISTINCT * FROM /Portfolios WHERE status = 'active'");
SelectResultsPtr resultsPtr = qry->execute(10);
SelectResultsIterator iter = resultsPtr->getIterator();
while (iter.hasNext()) {
    PortfolioPtr portfolio = dynCast<PortfolioPtr >(iter.next());
}
```
Querying the Portfolios Region

Examples show a sampling of specific queries that you can run against /portfolios on the server.
The query results for the data are listed in the previous table, Entry values in /portfolios. For the first several, the coding examples are included as well to show how you can execute the queries using the API.

Get distinct positions from portfolios with at least a $25.00 market value

This query assigns iterator variable names to the collections in the FROM clause. For example, the variable qryP is the iterator for the entry values in the /portfolios region. This variable is used in the second part of the FROM clause to access the values of the positions map for each entry value.

Query string:
```sql
SELECT DISTINCT posnVal
FROM /portfolios, positions.values posnVal TYPE Position
WHERE posnVal.mktValue >= 25.00
```
Results:
Collection of Position instances with secId: xxx, xxy, bbb, bbc

Retrieve all active portfolios

In the following example, a query response timeout parameter of 10 seconds is specified for the execute method to allow sufficient time for the operation to succeed.

Query string:
```sql
SELECT DISTINCT * FROM /portfolios WHERE status = 'active'
```
Results:
A collection of Portfolio objects for IDs 111, 222 and 333.

Code:
```c++
QueryServicePtr qrySvcPtr =
    cachePtr->getQueryService("examplePool");
QueryPtr qry = qrySvcPtr->newQuery("SELECT DISTINCT * FROM /Portfolios WHERE status = 'active'");
SelectResultsPtr resultsPtr = qry->execute(10);
SelectResultsIterator iter = resultsPtr->getIterator();
while(iter.hasNext())
{
    PortfolioPtr portfolio = dynCast<PortfolioPtr >(iter.next());
}
```

Retrieve all active portfolios that have type xyz

The type attribute is passed to the query engine in double quotes to distinguish it from the query keyword of the same name. A query response timeout parameter of 10 seconds is specified for the execute method to allow sufficient time for the operation to succeed.

Query string:
```sql
SELECT DISTINCT * FROM /portfolios
WHERE status = 'active' AND "type" = 'xyz'
```
Results:
A collection of Portfolio objects for IDs 111 and 222

Code:
```c++
QueryServicePtr qrySvcPtr =
```
Get the ID and status of all portfolios with positions in secId 'yyy'

Query string:
SELECT DISTINCT id, status FROM /portfolios
WHERE NOT (SELECT DISTINCT * FROM positions.values posnVal TYPE Position WHERE posnVal.secId='yyy').isEmpty

Results:
A collection of Struct instances, each containing an id field and a status field. For this data, the collection length is 1 and the Struct contains data from the entry with id 222.

Code:
QueryServicePtr qrySrvPtr = cachePtr->getQueryService("examplePool");
QueryPtr qry = qrySvcPtr->newQuery("import javaobject.Position; SELECT DISTINCT ID, status FROM "/Portfolios WHERE NOT (SELECT DISTINCT * FROM positions.values posnVal TYPE Position WHERE posnVal.secId='DELL').isEmpty");
SelectResultsPtr results = qry->execute(10);
SelectResultsIterator iter = results->getIterator();
while(iter.hasNext())
{
  Struct * si = (Struct*) iter.next().ptr();
  SerializablePtr id = si->operator[]("ID");
  SerializablePtr status = si->operator[]("status");
  printf("\nID=%s, status=%s", id->toString()->asChar(), status->toString()->asChar());
}

Modifying Cache Contents

To modify the cache based on information retrieved through querying, retrieve the entry keys and use them in the standard entry update methods.

The query service is a data access tool, so it does not provide any cache update functionality.

The next example shows entry key retrieval.

Get distinct entry keys and positions from active portfolios with at least a $25.00 market value

In the following example, retrieving the entry keys allows you to access the cached region entries for update. You cannot update the cache through the query engine.

Query string:
SELECT DISTINCT key, posnVal
FROM /portfolios.entrySet, value.positions.values posnVal TYPE Position
WHERE posnVal.mktValue >= 25.00

Results:
A SelectResults of Struct instances containing key, Position pairs.
Creating Indexes

An index can provide significant performance gains for query execution. You create and maintain indexes on the cache server.

A query run without an index iterates through every object in the collection on the cache server. If an index is available that matches part or all of the query specification, the query iterates only over the indexed set, and query processing time can be reduced.

When you create your indexes on the cache server, remember that indexes incur maintenance costs as they must be updated when the indexed data changes. An index that requires many updates and is not used very often may require more system resources than no index at all. Indexes also consume memory. For information on the amount of memory used for indexes, see the system configuration information in the Pivotal GemFire User's Guide.

You can create an index for remote querying declaratively on the cache server in a cache.xml file, as shown in the next example.

Creating an Index on a Cache Server Using a Server XML File

```xml
<region name="portfolios">
  <region-attributes . . . >
    <value-constraint>cacheRunner.Portfolio</value-constraint>
  </region-attributes>
  <index name="myFuncIndex">
    <functional from-clause="/portfolios" expression="status"/>
  </index>
  <index name="myPrimIndex">
    <primary-key field="id"/>
  </index>
  <entry> . . .
```

For detailed information about working with indexes configured on a cache server, see the section Working with Indexes in the GemFire User's Guide.
Remote Querying Requirements

Note the particular requirements for using region endpoints; setting server region data policy and scope; implementing equals and hashcode methods; and setting object type constraints.

- Using Region Endpoints
- Setting Server Region Data Policy and Scope
- Implementing the equals and haschcode Methods
- Setting Object Type Constraints

Using Region Endpoints

When you are using region endpoints, at least one region must exist on the native client before a query can be executed through the client. All objects in the region belong to the same class hierarchy (homogenous types).

Setting Server Region Data Policy and Scope

Native client remote querying only accesses the data that is available in the remote cache server region, so no local cache loading operations are performed. Depending on the cache server region's scope and data-policy attribute settings, this could mean that your queries and indexes only see a part of the data available for the server region in the distributed cache.

To ensure a complete data set for your queries and indexes, your cache server region must use one of the REPLICATE region shortcut settings in the region attribute refid or it must explicitly have its data policy set to replicate or persistent-replicate.

For a cache server region, setting its data policy to replicate or persistent-replicate ensures that it reflects the state of the entire distributed region. Without replication, some server cache entries may not be available.

Depending on your use of the server cache, the non-global distributed scopes distributed-ack and distributed-no-ack may encounter race conditions during entry distribution that cause the data set to be out of sync with the distributed region. The global scope guarantees data consistency across the distributed system, but at the cost of reduced performance.

The following table summarizes the effects of cache server region scope and data policy settings on the data available to your querying and indexing operations. For more information, see the Distributed and Replicated Regions in the Pivotal GemFire User's Guide.

Table 21: Effects of Cache Server Region Scope and Data Policy on the Data Available for Querying

<table>
<thead>
<tr>
<th>Region Scope</th>
<th>Not replicated</th>
<th>Replicated</th>
</tr>
</thead>
<tbody>
<tr>
<td>distributed-ack or distributed-no-ack</td>
<td>N/A</td>
<td>FULL data set (if no race conditions).</td>
</tr>
<tr>
<td>global</td>
<td>N/A</td>
<td>FULL data set.</td>
</tr>
</tbody>
</table>

Implementing the equals and haschcode Methods

The Portfolio and Position query objects for the cache server must have the equals and hashCode methods implemented, and those methods must provide the properties and behavior mentioned in the online documentation for Object.equals and Object.hashCode. Inconsistent query results can occur if these methods are absent.
See the Object class description in the GemFire online Java API documentation for more information about the equals and hashCode methods.

**Setting Object Type Constraints**

Performing queries on cache server regions containing heterogeneous objects, which are objects of different data types, may produce undesirable results. Queries should be performed only on regions that contain homogeneous objects of the same object type, although subtypes are allowed.

So your queries will address homogeneous data types, you need to be aware of the values that the client adds to the server. You can set the key-constraint and value-constraint region attributes to restrict region entry keys and values to a specific object type. However, because objects put from the client remain in serialized form in the server cache and do not get deserialized until a query is executed, it is still possible to put heterogeneous objects from the client.

See the vFabric GemFire User's Guide for descriptions of the key-constraint and value-constraint attributes for the cache server. See *Specifying the object types of FROM clause collections* for more information on associating object types with queries.

**Using Query Strings in the Native Client**

To use a query string in a native client, specify the string as a parameter in a QueryService::newQuery method, then execute the query using Query::execute, passing in the required parameters.

Alternatively, if an expression evaluates to a boolean value, you can specify it using the region shortcut methods Region::existsValue, Region::selectValue, and Region::query. These shortcut methods evaluate whether given expressions return any entries and return a single value entry, respectively. See *Region Shortcut Query Methods* for more information about these shortcut methods.

If your query requires any IMPORT statements, you must include these before the SELECT statement in the query string that is passed to the query engine. It should be a fully qualified package name relative to the cache server. The Java class definition must exist and have the exact footprint as the native client C++ class.
FROM Clause

The **FROM** clause establishes collections of objects that are iterated over by the remainder of the query.

The attributes of the objects in these collections are added to the name space scope for the remainder of
the **FROM** clause as well as for the **WHERE** clause and the **SELECT** projection list.

Each **FROM** clause expression must evaluate to a collection. The expression `/portfolios.keySet` is valid
because it evaluates to a **Collection**, but `/portfolios.name`, which evaluates to a **String**, causes an
exception to be thrown.

Like the SQL query, which iterates over the tables named in its **FROM** clause, the **OQL** query iterates over
the **Collections** established in its **FROM** clause.

In the following query, `positions.values` evaluates to a **Collection** because `positions` is a **Map**, and
the method `values` on **Map** returns a **Collection**.

```java
IMPORT javaobject.Position;
SELECT DISTINCT "type"
FROM /portfolios, positions.values posnVal TYPE Position
WHERE posnVal.qty > 1000.00
```

Every expression in the **FROM** clause must evaluate to a **Collection**. For a **Map**, the method `values`
returns a **Collection**.

If `positions` were a List instead of a **Map**, this query could be used to retrieve the data:

```java
IMPORT javaobject.Position;
SELECT DISTINCT "type"
FROM /portfolios, positions posnVal TYPE Position
WHERE posnVal.qty >= 1000.00
```

A List is a **Collection**, so you can access it directly or through its `toArray` method.

For each object type accessed in your **FROM** clause, use the method that returns a **Collection** for that
object.

Each expression in the **FROM** clause can be any expression that evaluates to a **Collection**. An expression
in the **FROM** clause is typically a path expression that resolves to a region in the cache so that the values in
the region become the collection of objects to filter.

For example, this is a simple **SELECT** statement that evaluates to a set of all the entry value objects of the
region `/portfolios` with active status. The collection of entry values provided by the **FROM** clause is
traversed by the **WHERE** clause, which accesses each element’s status attribute for comparison.

```java
SELECT DISTINCT * FROM /portfolios WHERE status = 'active'
```

If the **FROM** clause has only one expression in it, the result of the clause is the single collection that the
expression evaluates to. If the clause has more than one expression in it, the result is a collection of
structs that contain a member for each of those collection expressions. For example, if the **FROM** clause
contains three expressions that evaluate to collections `C1`, `C2`, and `C3`, the **FROM** clause generates a set of
struct(`x1`, `x2`, `x3`) where `x1`, `x2`, and `x3` represent nested iterations over the collections specified.

If the collections are independent of each other, this struct represents their cartesian product.

In this query, the **FROM** clause produces a struct of `portfolio` and position pairs to be iterated. Each
element in the struct contains the portfolio and one of its contained positions.

```java
IMPORT javaobject.Position;
SELECT DISTINCT "type" FROM /portfolios, positions TYPE Position
WHERE qty > 1000.00
```
To understand the effects of `FROM` expressions on query scope, see *Drilling Down for Modifying Query Scope*.

**Using Iterator Variables**

For each collection expressed in the `FROM` clause, you can associate an explicit variable. The variable is added to the current scope and becomes the iterator variable bound to the elements of the collection as they are iterated over. In this example, `pflo` and `posnVal` are both explicit iterator variables.

**Query Using Explicit Iterator Variables**

```java
IMPORT javaobject.Position;
SELECT DISTINCT pflo."type", posnVal.qty
FROM /portfolios pflo, positions.values posnVal TYPE Position
WHERE pflo.status = 'active' and posnVal.mktValue > 25.00
```

**Importing and Using Object Classes**

To facilitate the specification of type in variable type declarations and in typecasting expressions, a query string can have `IMPORT` statements preceding the declarations. By using `IMPORT` in the query string, the client can tell the cache server about the class definition of the serialized object that is present in the cache server region.

The only place you can have a package name in a query is in an import statement. These are valid:

```java
IMPORT com.myFolder.Portfolio;
IMPORT com.myFolder.Portfolio AS MyPortfolio;
```

The first form of the import statement allows `Portfolio` to be used as the name of the class, `com.myFolder.Portfolio`. The second form provides an alternative class name, `MyPortfolio`, to be used. This is useful when a class name is not unique across packages and classes in a single query.

**Using Imported Classes**

The following example uses imported classes:

```java
IMPORT com.commonFolder.Portfolio;
IMPORT com.myFolder.Portfolio AS MyPortfolio;
SELECT DISTINCT mpflo.status
FROM /portfolios pflo TYPE Portfolio,
/myPortfolios mpflo TYPE MyPortfolio,
WHERE pflo.status = 'active' and mpflo.id = pflo.id
```

This entire query string must be passed to the query engine, including the `IMPORT` statements.

Common type names do not require an `IMPORT` statement. The following table lists the types that are defined by the system and the Java types they represent.

**Predefined Class Types**

The `FROM` clause establishes collections of objects that are iterated over by the remainder of the query. The attributes of the objects in these collections are added to the name space scope for the remainder of the `FROM` clause as well as for the `WHERE` clause and the `SELECT` projection list.

The type specification can be an imported type or any of these predefined types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Java</th>
<th>C++</th>
<th>.NET</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>short</td>
<td>CacheableInt16</td>
<td>Int16</td>
</tr>
</tbody>
</table>
### Specifying the Object Types of FROM Clause Collections

To resolve implicit attribute names, the query engine must be able to associate each attribute or method name to a single iterator expression in the **FROM** clause.

Depending on the complexity of the query, the engine may be able to discover the proper associations on its own, but providing the specifications described here increases the chances for success.

The server region being queried should contain only homogeneous objects of the same type. See **Setting Object Type Constraints** for more information.

The object type information must be available when the query is created. To provide the appropriate information to the query engine, specify the type for each of your **FROM** clause collection objects by importing the object's class before running the query and typecasting the object inside the query. For the example region, this query is valid (all of the examples in this chapter assume that this **IMPORT** statement is provided):  

**Query Using IMPORT and TYPE for Object Typing**

```java
IMPORT javaobject.Position;
SELECT DISTINCT mktValue
FROM /portfolios, positions.values TYPE Position
WHERE mktValue > 25.00
```

This entire query string must be passed to the query engine, including the **IMPORT** statement. Import the object's class before running the query and typecast the object inside the query. For the example region, both of these queries are valid:

**Query Using IMPORT and Typecasting for Object Typing**

```java
IMPORT javaobject.Position;
SELECT DISTINCT value.mktValue
FROM /portfolios, (map<string,Position>)positions
WHERE value.mktValue > 25.00
IMPORT cacheRunner.Position;
```
This entire query string must be passed to the query engine, including the `IMPORT` statement. Use named iterators in the `FROM` clause and explicitly prefix the path expression with iterator names.

**Query Using Named Iterators for Object Typing**

```
SELECT DISTINCT mktValue
    FROM /portfolios, {collection<Position>}positions.values
    WHERE mktValue > 25.00
```

```
SELECT DISTINCT posnVal
    FROM /portfolios pflo, pflo.positions.values posnVal
    WHERE posnVal.mktValue >= 25.00
```

The `IMPORT` statements in these examples assume that `$GEMFIRE/examples/dist/classes` is in the `CLASSPATH`. This is required so the cache server can process `IMPORT` statements. The class’s package name cannot be used in the `FROM` clause. The package name must be specified in an `IMPORT` statement.

There is one exception to these typing guidelines. If one `FROM` expression lacks explicit typing, the query engine associates all unresolved attributes with that expression and creates the query. An exception is thrown if any of these attributes are not found at execution time.
**SELECT Projection List**

The projections in the SELECT projection list are used to transform the results of the WHERE search operation.

You specify the projection list either as * or as a comma delimited list of expressions. For *, the interim results of the WHERE clause are returned from the query. Otherwise, the set of objects in the interim results are iterated and the projections applied to each of the objects. During the application of the projection list, the attributes of the objects being traversed are in scope for name resolution.

You can also specify retrieval of the entry keys in your projection list. This allows you to access the associated cached entries for modification and other purposes. The following example shows how the Region entry key can be obtained by using the region entries in the FROM clause and using appropriate projections. This query runs on the /portfolios region, returning a set of `struct<key:string, id:string, secId:string>` where `key` is the key of the region entry, `id` is an entry ID, and `secId` is a secId of a `positionsmap` for the entry.

```sql
SELECT DISTINCT key, entry.value.id, posnVal.secId
FROM /portfolios.entrySet entry, entry.value.positions.values posnVal
WHERE entry.value."type" = 'xyz' AND posnVal.secId = 'XXX'
```
SELECT Statement Query Results

The result of a SELECT statement is a collection that implements the SelectResults interface or it is UNDEFINED.

The SelectResults returned from the SELECT statement is either a collection of objects or a Struct collection containing the objects. (See also the online API documentation for Query.)

Because a SELECT statement returns a result, it can be composed with other expressions like the following example:

```java
(SELECT DISTINCT * FROM /portfolios WHERE status = 'active').iterator
```

A collection of objects is returned in two cases:

- When only one expression is specified by the projection list and that expression is not explicitly specified using the `fieldname:expression` syntax
- When the SELECT list is `*` and a single collection is specified in the FROM clause

<table>
<thead>
<tr>
<th>SELECT FROM</th>
<th>Single Expressions</th>
<th>Multiple Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>single expression</td>
<td>Objects</td>
<td>Objects. (Struct if the projection specifies a field name.)</td>
</tr>
<tr>
<td>multiple expressions</td>
<td>Struct</td>
<td>Struct</td>
</tr>
</tbody>
</table>

When a Struct is returned, the name of each field in the Struct is determined as follows:

- If a field is specified explicitly using the `fieldname:expression` syntax, the fieldname is used.
- If the SELECT projection list is `*` and an explicit iterator expression is used in the FROM clause, the iterator variable name is used as the field name.
- If the field is associated with a region or attribute path expression, the last attribute name in the expression is used.

If names can not be decided based on these rules, arbitrary unique names are generated by the query processor.

These examples show how the projections and FROM clause expressions are applied.

<p>| SELECT &lt;*&gt; FROM &lt;single expression&gt; | SELECT DISTINCT * FROM /portfolios WHERE status = 'active' | Returns the Collection of active portfolios objects. |</p>
<table>
<thead>
<tr>
<th>SQL Query</th>
<th>Java Object Import</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SELECT &lt;single expression&gt; FROM &lt;multiple expression&gt; (without fieldName mentioned)</code></td>
<td><code>IMPORT javaobject.Position; SELECT DISTINCT secId FROM /portfolios, positions.values TYPE Position WHERE status = 'active'</code></td>
<td>Returns the Collection of secIds (CacheableString objects) from the positions of active portfolios.</td>
</tr>
<tr>
<td><code>SELECT &lt;single expression&gt; FROM &lt;multiple expression&gt; (with fieldName mentioned)</code></td>
<td><code>IMPORT javaobject.Position; SELECT DISTINCT secIdField:secId FROM /portfolios, positions.values TYPE Position WHERE status = 'active'</code></td>
<td>Returns struct&lt;secIdField: CacheableString&gt; for the active portfolios. (Compare to the results for the prior query.)</td>
</tr>
<tr>
<td><code>SELECT &lt;*&gt; FROM &lt;multiple expression&gt;</code></td>
<td><code>IMPORT javaobject.Position; SELECT DISTINCT * FROM /portfolios, positions.values TYPE Position WHERE status = 'active'</code></td>
<td>Returns a Collection of struct&lt;portfolios: Portfolio, values: Position&gt; for the active portfolios.</td>
</tr>
</tbody>
</table>
**WHERE Clause**

The optional WHERE clause defines the search criteria for the selection, filtering the set of elements specified by the FROM clause.

Without a WHERE clause, the SELECT projection list receives the entire collection or set of collections as specified in the FROM clause.

The query processor searches the collection for elements that match the conditions specified in the WHERE clause conditions. If there is an index on an expression matched by the WHERE clause, then the query processor may use the index to optimize the search and avoid iterating over the entire collection. For more information on indexes, see the *Advanced Querying* chapter in the GemFire User's Guide.

A WHERE clause expression is a boolean condition that is evaluated for each element in the collection. If the expression evaluates to true for an element, the query processor passes that element on to the SELECT projection list. This example uses the WHERE clause to return the portfolio objects in the region that have a type *xyz*.

```
SELECT DISTINCT * FROM /portfolios WHERE "type" = 'xyz'
```

The next query returns the set of all portfolios with a type of *xyz* and active status.

```
SELECT DISTINCT * FROM /portfolios WHERE "type" = 'xyz' AND status = 'active'
```
Joins

If collections in the FROM clause are not related to each other, you can use the WHERE clause to join them.

The statement below returns all the persons from the /Persons region with the same name as a flower in the /Flowers region.

```
SELECT DISTINCT p FROM /Persons p, /Flowers f WHERE p.name = f.name
```

Indexes are supported for region joins. To create indexes for region joins, you create single-region indexes for both sides of the join condition. These are used during query execution for the join condition.

Accessing Cached Data

Accessing your cached data through the querying service is similar to accessing database contents through SQL queries. How you specify your regions and region contents is particular to the native client.

The query language supports drilling down into nested object structures. Regions can contain nested data collections that are unavailable until referenced in the FROM clause.

This discussion describes how to navigate to your cached data through the native client query service.

**Note:** Querying and indexing only operate on remote cache server contents.
Basic Region Access

In the context of a query, you specify the name of a region by its full path, starting with a forward slash (/).

Object Attributes

You can access the Region object's public fields and methods from a region path, referred to as the region's attributes. Using this method, `/portfolios.name` returns "portfolios" and `/portfolios.name.length` returns 10. An attribute is mapped to a Java class member in three possible ways with the following priority until a match is found. If the attribute is named x, then:

```
public method getX()
public method x()
public field x
```

Note: The term attribute in this context is not the same as a region attribute.

Region Data

You can also access entry keys and entry data through the region:

- `/portfolios.keySet` returns the Set of entry keys in the region
- `/portfolios.entrySet` returns the Set of Region.Entry objects
- `/portfolios.values` returns the Collection of entry values
- `/portfolios` return the Collection of entry values

Note: These collections are immutable. Invoking modifier methods on them, such as add and remove, result in an UnsupportedOperationException.

For the last two bullets, the FROM clause `/portfolios.values` and `/portfolios` return the same thing.
Attribute Visibility

Within the current query scope, you can access any available object or object attribute.

In querying, an object's attribute is any identifier that can be mapped to a public field or method in the object.

In the FROM specification, any object that is in scope is valid, so at the beginning of a query all cached regions and their attributes on the cache server are in scope.

This query is valid because name resolves to the Region method `getName`:

```
/portfolios.name
```

This query is valid because `toArray` resolves to the Collection method with the same name:

```
SELECT DISTINCT * FROM /portfolios.toArray
```

You cannot, however, refer to the attribute of a collection object in the region path expression where the collection itself is specified. The following statement is invalid because neither Collection nor Region contain an attribute named positions. The entry values collection (specified by `/portfolios`) that does contain an attribute named positions is not yet part of the query name space.

```
/* INCORRECT: positions is not an attribute of Region or of Collection */
SELECT DISTINCT * FROM /portfolios.positions
```

The following SELECT statement is valid because positions is an element of the entry value collection that is specified by `/portfolios`. The entry value collection is in scope as soon as the specification in the FROM expression is complete (before WHERE or SELECT are evaluated).

```
SELECT DISTINCT positions FROM /portfolios
```

You can also refer to positions inside the FROM clause after the `/portfolios` entry value collection is created. In this example, positions is an element of the `/portfolios` entry value collection and values is an attribute of positions:

```
IMPORT javaobject.Position;
SELECT DISTINCT posnVal
FROM /portfolios, positions.values posnVal TYPE Position
WHERE posnVal.mktValue >= 25.00
```

After the comma in the FROM clause, `/portfolios` is in scope, so its value collection can be iterated. In this case, this is done with the second FROM clause specification, `positions.values`. 
Modifying Query Scope

The query engine resolves names and path expressions according to the name space that is currently in scope in the query. This is not the region scope attribute, but the scope of the query statement.

The initial name space for any query is composed of the region paths of the cache on the cache server and the attributes of those paths. New name spaces are brought into scope based on the FROM clause in the SELECT statement. For example, in this query the FROM expression evaluates to the collection of entry values in /portfolios. This is added to the initial scope of the query and status is resolved within the new scope.

```
SELECT DISTINCT *
FROM /portfolios
WHERE status = 'active'
```

Each FROM clause expression must resolve to a collection of objects available for iteration in the query expressions that follow. In the example above, /portfolios resolves to the Collection of entry values in the region. The entry value collection is iterated by the WHERE clause, comparing the status field to the string active. When a match is found, the value object is added to the return set.

In the following query, the collection specified in the first FROM clause expression is used by the second FROM clause expression and by the projections of the SELECT statement.

```
IMPORT cacheRunner.Position;
SELECT DISTINCT "type"
FROM /portfolios, positions.values posnVal TYPE Position
WHERE posnVal.qty > 1000.00
```

**Note:** You cannot change the order of the expressions in this FROM clause. The second expression depends on the scope created by the first expression.
Nested Query Scopes

You can nest scopes by using nested `SELECT` statements. Names in an inner scope hide identical names in an outer scope.

In the query below, the inner `SELECT` creates a new scope, the positions of the current portfolio, inside the outer `SELECT`’s scope, `/portfolios`. This inner scope (the collection of entry values from the `/portfolios` region) is first searched for the `secId` element. The outer scope is searched only if the `secId` element is not found in the inner scope.

```javaobject
IMPORT javaobject.Position;
SELECT DISTINCT * FROM /portfolios
WHERE NOT
(SELECT DISTINCT * FROM positions.values TYPE Position
    WHERE secId='YYY').isEmpty
```

This statement shows the outer scope in bold. The outer scope has all the attributes of a Portfolio in it.

```javaobject
IMPORT javaobject.Position;
SELECT DISTINCT * FROM /portfolios
WHERE NOT
(SELECT DISTINCT * FROM positions.values TYPE Position
    WHERE secId='YYY').isEmpty
```

Now the statement with the inner scope is shown in bold. The inner scope has all the attributes of a Portfolio in it (inherited from the outer scope), and all the attributes of a Position as well.

```javaobject
IMPORT javaobject.Position;
SELECT DISTINCT * FROM /portfolios
WHERE NOT
(SELECT DISTINCT * FROM positions.values TYPE Position
    WHERE secId='YYY').isEmpty
```
When Names Cannot Be Resolved

When a query is executed and a name or path expression resolves to more than one region name in the scope, or if the name cannot be resolved at all, the client receives a `QueryException`. The `QueryException` contains the message that is generated for the exception that occurs on the server.

Query Language Elements

This section discusses various aspects and tools of the native client query engine.
**Method Invocation**

The query language supports method invocation inside query expressions.

The query processor maps attributes in query strings using the attribute rules described in *Object attributes*. Methods declared to return `void` evaluate to `null` when invoked through the query processor.

If you know that the attribute name maps to a public method that takes no parameters, you can simply include the method name in the query string as an attribute. For example, `emps.isEmpty` is equivalent to `emps.isEmpty()`. In the following example, the query invokes `isEmpty` on positions, and returns the set of all portfolios with no positions.

```
SELECT DISTINCT * FROM /portfolios WHERE positions.isEmpty
```

The native client also supports the invocation of public methods with parameters. To invoke methods with parameters, include the method name as an attribute in the query string and provide the method arguments between parentheses. You can only use constants in the query strings.
Query Language Literals Supported by Native Client

Query language expressions can contain literals as well as operators and attribute names. The native client supports many types of literals.

- **boolean**. Boolean value, either `TRUE` or `FALSE`.
- **integer** and **long**. Type `long` if it is suffixed with the ASCII letter L. Otherwise it is of type `int`.
- **floating point**. Type `float` if it is suffixed with an ASCII letter F. Otherwise its type is double and it can optionally be suffixed with an ASCII letter D. A double or floating point literal can optionally include an exponent suffix of E or e, followed by a signed or unsigned number.
- **string**. Delimited by single quotation marks. Embedded single quotation marks are doubled. For example, the character string `'Hello'` evaluates to the value `Hello`, while the character string `'He said, ''Hello'''` evaluates to `He said, 'Hello'`. Embedded newlines are kept as part of the string literal.
- **char**. Type `char` if it is a string literal prefixed by the keyword `CHAR`; otherwise it is of type `string`. The `CHAR` literal for the single quotation mark character is `CHAR '''` (four single quotation marks).
- **date**. `java.sql.Date` object that uses the JDBC format prefixed with the `DATE` keyword: `DATE yyyy-mm-dd`. In the `Date`, `yyyy` represents the year, `mm` represents the month, and `dd` represents the day. The year must be represented by four digits; a two-digit shorthand for the year is not allowed.
- **time**. Not supported.
- **timestamp**. Not supported.
- **NIL**. Equivalent alternative of `NULL`.
- **NULL**. Same as null in Java.
- **UNDEFINED**. Special literal that is a valid value for any data type. An `UNDEFINED` value is the result of accessing an attribute of a null-valued attribute. If you access an attribute that has an explicit value of null, then it is not undefined. For example, if a query accesses the attribute `address.city` and `address` is null, then the result is undefined. If the query accesses `address`, then the result is not undefined, it is null.
Type Conversions

Java rules within a query string require the query processor to perform implicit conversions and promotions under certain cases in order to evaluate expressions that contain different types.

The query processor performs binary numeric promotion, method invocation conversion, and temporal type conversion.

- Binary numeric promotion
- Method invocation conversion
- Temporal type conversion

Binary numeric promotion

Binary numeric promotion widens all operands in a numeric expression to the widest representation used by any of the operands. In each expression, the query processor applies the following rules in order:

- If either operand is of type double, the other is converted to double.
- If either operand is of type float, the other is converted to float.
- If either operand is of type long, the other is converted to long.
- Both operands are converted to type int.

The query processor performs binary numeric promotion on the operands of the following operators:

- comparison operators <, <=, >, and >=
- equality operators = and <>

This is essentially the same behavior as in Java, except that chars are not considered to be numeric in the native client query language.

Method invocation conversion

Method invocation conversion in the query language follows the same rules as Java method invocation conversion, except that the query language uses runtime types instead of compile time types, and handles null arguments differently than in Java. One aspect of using runtime types is that an argument with a null value has no typing information, and so can be matched with any type parameter. When a null argument is used, if the query processor cannot determine the proper method to invoke based on the non-null arguments, it throws an AmbiguousNameException. For more information on method invocation in query strings, see Method Invocation.

Temporal type conversion

The temporal types that the query language supports on the cache server include the Java types java.util.Date and java.sql.Date, which are treated the same and can be freely compared and used in indexes. When compared with each other, these types are all treated as nanosecond quantities.

Remote Query API

You use the native client querying API to access all the querying functionality discussed in the previous sections.

This chapter gives a general overview of the interfaces and classes that are provided by the Query package API, and the shortcut methods provided in the Region interface. For complete, current information on the classes and interfaces discussed here, see the native client online API documentation.
Creating and Managing Queries

You create queries on the cache server by obtaining a QueryService method and manage them through the resulting Query object. The Region interface has several shortcut query methods.

The newQuery method for the Query interface binds a query string. By invoking the execute method, the query is submitted to the cache server and returns SelectResults, which is either a ResultSet or a StructSet.

The QueryService method is the entry point to the query package. It is retrieved from the Cache instance through Cache::getQueryService. If you are using the Pool API you must obtain the QueryService from the pools and not from the cache.

Query

A Query is obtained from a QueryService method, which is obtained from the cache. The Query interface provides methods for managing the compilation and execution of queries, and for retrieving an existing query string.

You must obtain a Query object for each new query. The following example demonstrates the method used to obtain a new instance of Query:

```cpp
QueryPtr newQuery(const char * querystr);
```

Region Shortcut Query Methods

The Region interface has several shortcut query methods. All take a query predicate which is used in the WHERE clause of a standard query. See WHERE Clause for more information. Each of the following examples also set the query response timeout to 10 seconds to allow sufficient time for the operation to succeed.

- The query method retrieves a collection of values satisfying the query predicate. This call retrieves active portfolios, which in the sample data are the portfolios with keys 111, 222, and 333:

```cpp
SelectResultsPtr results = regionPtr->query("status 'active' ");
```

- The selectValue method retrieves one value object. In this call, you request the portfolio with ID ABC-1:

```cpp
SerializablePtr port = region->selectValue("ID='ABC-1' ");
```

- The existsValue method returns a boolean indicating if any entry exists that satisfies the predicate. This call returns false because there is no entry with the indicated type:

```cpp
bool entryExists = region->existsValue("'type' = 'QQQ' ");
```

For more information about these shortcut query methods, see the Region class description in the native client online API documentation.
Query Result Sets

- **SelectResults.** Executes the query on the cache server and returns the results as either a ResultSet or a StructSet.
- **SelectResultsIterator.** Iterates over the items available in a ResultSet or StructSet.
- **ResultSet.** Obtained after executing a Query, which is obtained from a QueryService that is obtained from a Cache class.
- **StructSet.** Used when a SELECT statement returns more than one set of results. This is accompanied by a Struct, which provides the StructSet definition and contains its field values.
Query Code Samples Returning ResultSet

API examples demonstrate methods for returning ResultSet for both built-in and user-defined data types.

Query Returns a ResultSet for a Built-In Data Type

```cpp
QueryServicePtr qrySvcPtr = cachePtr->getQueryService("examplePool");
QueryPtr query =
    qrySvcPtr->newQuery("select distinct pkid from /Portfolios");
//specify 10 seconds for the query timeout period
SelectResultsPtr results = query->execute(10);
if (results == NULLPTR)
    {
        printf("\nNo results returned from the server");
    }

//obtaining a handle to resultset
ResultSetPtr rs(dynamic_cast<ResultSet*> (results.ptr()));
if (rs == NULLPTR)
    {
        printf("\nResultSet is not obtained \n"); return;
    }
//iterating through the resultset using row index.
for (int32_t row=0; row < rs->size(); row++)
    {
        SerializablePtr ser(*rs)[row]);
        CacheableStringPtr str(dynamic_cast<CacheableString*> (ser.ptr()));
        if (str != NULLPTR)
            {
                printf("\n string column contains - %s \n", str->asChar() );
            }
    }
```

Query Returns a ResultSet for a User-Defined Data Type

```cpp
QueryServicePtr qrySvcPtr = cachePtr->getQueryService("examplePool");
const char * querystring = "select distinct * from /Portfolios";
QueryPtr query = qrySvcPtr->newQuery(querystring);
//specify 10 seconds for the query timeout period
SelectResultsPtr results = query->execute(10);
if (results == NULLPTR)
    {
        printf("\nNo results returned from the server");
    }

//obtaining a handle to resultset
ResultSetPtr rs(dynamic_cast<ResultSet*> (results.ptr()));
if (rs == NULLPTR)
    {
        printf("\nResultSet is not obtained \n"); return;
    }
//iterating through the resultset using iterators.
SelectResultsIterator iter = rs->getIterator();
while (iter.hasNext())
    {
        SerializablePtr ser = iter.next();
        PortfolioPtr port(dynamic_cast<Portfolio*> (ser.ptr()));
        if (port != NULLPTR)
            {
                printf("\nPortfolio object is - %s \n", port->toString()->asChar() );
            }
    }
    // end of rows
```
Query Code Samples Returning StructSet

These examples return a `StructSet` for built-in and user-defined data types, `Struct` objects, and collections.

### Query Returning a StructSet for a Built-In Data Type

```c
QueryServicePtr qrySvcPtr = cachePtr->getQueryService("examplePool");
const char * querystring = "SELECT DISTINCT ID, pkid, status, getType FROM /Portfolios";
QueryPtr query = qrySvcPtr->newQuery(querystring);
//specify 10 seconds for the query timeout period
SelectResultsPtr results = query->execute(10);
if (results == NULLPTR) {
    printf("\nNo results returned from the server\n");
}
//obtaining a handle to resultset
StructSetPtr ss(dynamic_cast<StructSet*>(results.ptr()));
if (ss == NULLPTR) {
    printf("\nStructSet is not obtained \n");
    return;
}
//iterating through the resultset using indexes.
for ( int32_t row=0; row < ss->size(); row++) {
    Struct * siptr = (Struct*) dynamic_cast<Struct*> ( ((*ss)[row]).ptr() );
    if (siptr == NULL) {
        printf("\nstruct is empty \n");
        continue;
    }
    //iterate through fields now
    for( int32_t field=0; field < siptr->length(); field++) {
        SerializablePtr fieldptr(*siptr)[field]);
        if(fieldptr == NULLPTR ) {
            printf("\nnull data received\n");
        }
        CacheableStringPtr
            str(dynamic_cast<CacheableString*>(fieldptr.ptr()));
        if (str == NULLPTR) {
            printf("\n field is of some other type \n");
        } else {
            printf("\n Data for %s is %s ", siptr->getFieldName(field),
               str->asChar() );
        }
    } //end of columns
} // end of rows
```

### Returning Struct Objects

```c
QueryServicePtr qrySvcPtr = cachePtr->getQueryService("examplePool");
const char * querystring = "SELECT DISTINCT derivedProjAttrbts, key: p.key FROM 
```
/Portfolios.entries p, (SELECT DISTINCT x.ID, myPos.secId FROM "/Portfolios x, x.positions.values AS myPos) derivedProjAttrbts WHERE 
"p.value.ID = derivedProjAttrbts.ID AND derivedProjAttrbts.secId = 'IBM'";
QueryPtr query = qrySvcPtr->newQuery(querystring);
//specify 10 seconds for the query timeout period
SelectResultsPtr results = query->execute(10);
if (results == NULLPTR)
{
    printf( "\nNo results returned from the server\n");
}
//obtaining a handle to resultset
StructSetPtr ss(dynamic_cast<StructSet*> (results.ptr()));
if (ss == NULLPTR)
{
    printf ("\nStructSet is not obtained \n");
    return;
}
//iterating through the resultset using indexes.
for ( int32_t row=0; row < ss->size(); row++)
{
    Struct * siptr = (Struct*) dynamic_cast<Struct*> ( (*(ss)[row]).ptr() );
    if (siptr == NULL) { printf("\nstruct is empty \n"); }
    //iterate through fields now
    for( int32_t field=0; field < siptr->length(); field++) {
        SerializablePtr fieldptr((*siptr)[field]);
        if(fieldptr == NULLPTR )
        {
            printf("\nnull data received\n");
        }
        else
        {
            StructPtr simpl(dynamic_cast<Struct*> (fieldptr.ptr()));
            if (simpl == NULLPTR)
            {
                printf("\n field is of some other type \n"); continue;
            }
            printf( "\n struct received %s \n", siptr->getFieldName(field),
                    str->asChar() );
            for( int32_t inner_field=0; inner_field < simpl->length(); inner_field++)
            {
                SerializablePtr innerfieldptr((*simpl)[inner_field]);
                if (innerfieldptr == NULLPTR)
                {
                    printf("\ninnerfield is NULL\n");
                }
                else
                {
                    DateFormatPtr innerfieldstr(dynamic_cast<DateFormat*>(innerfieldptr.ptr()));
                    if (innerfieldstr == NULLPTR)
                    {
                        printf("\n Innerfield str is NULL\n");
                    }
                    else
                    {
                        printf("\n Data for %s is %s ",
                                simpl->getFieldName(inner_field),
                                str->asChar() );
                    }
                }
            }
        }
}
} //end of columns
Returning Collections

QueryServicePtr qrySvcPtr = cachePtr->getQueryService("examplePool");
const char * querystring = "select distinct ID, names from /Portfolios";
QueryPtr query = qrySvcPtr->newQuery(querystring);
SelectResultsPtr results = query->execute(10);
if (results == NULLPTR) {
    printf("\nNo results returned from the server\n");
}
//obtaining a handle to resultset
StructSetPtr ss(dynamic_cast<StructSet*>(results.ptr()));
if (ss == NULLPTR) {
    printf ("\nStructSet is not obtained \n");
    return;
}
//iterating through the resultset using indexes.
for ( int32_t row=0; row < ss->size(); row++) {
    Struct * siptr = dynamic_cast<Struct*>( (*ss)[row]).ptr() );
    if (siptr == NULL) {
        printf("\nstruct is empty \n");
        continue;
    }
    //iterate through fields now
    for( int32_t field=0; field < siptr->length(); field++){
        SerializablePtr fieldptr(*siptr)[field]);
        if(fieldptr == NULLPTR ) {
            printf("\nnull data received\n");
        }
        CacheableStringPtr
        str(dynamic_cast<CacheableString*>(fieldptr.ptr()));
        if (str != NULLPTR) {
            printf ("\n Data for %s is %s ", siptr->getFieldName(field),
                     str->asChar() );
        }else {
            CacheableObjectArrayPtr
            coa(dynamic_cast<CacheableObjectArray*>(fieldptr.ptr()));
            if (coa == NULLPTR) {
                printf("\n field is of some other type\n"); continue;
            }
            printf( "\n objectArray received %s \n",
                    siptr->getFieldName(field) );
            for(unsigned arrlen=0; arrlen < (uint32_t)coa->length();
                 arrlen++)
                printf("\n Data for %s is %s ",siptr->getFieldName(field),
                        coa->operator[](arrlen)->toString()->asChar());
        }
    } //end of columns
} //end of rows
Continuous Querying

Continuous Querying describes how to implement continuous querying in the Pivotal GemFire native client so that C++ and C# .NET clients can run queries against events in the GemFire cache server region. It also describes main features and the native client CQ API.

How Continuous Querying Works

C++ and C# .NET clients register interest in events using simple query expressions. Events are sent to client listeners that you can program to do whatever your application requires.

- Overview of CQ Operations
- Logical Architecture and Data Flow
- State and Life Cycle

Overview of CQ Operations

You subscribe to server-side events using SQL-type query filtering. The native client sends a query to the server side for execution and receives the events that satisfy the criteria.

For example, in a region storing stock market trade orders, you can retrieve all orders over a certain price by running a CQ with a query like this:

```
SELECT * FROM /tradeOrder t WHERE t.price > 100.00
```

When the CQ is running, the server sends the client all new events that affect the results of the query. On the native client side, listeners programmed by you receive and process incoming events. For the example query on /tradeOrder, you might program a listener to push events to a GUI where higher-priced orders are displayed. CQ event delivery uses the client/server subscription framework described in Client to Server Connection Process.

CQs do not update the native client region. This is in contrast to other server-to-client messaging, like the updates sent to satisfy interest registration and responses to get requests from the client. CQs are notification tools for the CQ listeners, which can be programmed in any way your application requires.

When a CQ is running against a server region, each entry event is evaluated against the CQ query by the thread that updates the server cache. If either the old or the new entry value satisfies the query, the thread puts a CqEvent in the client's queue. The CqEvent contains information from the original cache event, plus information specific to the CQ's execution. Once received by the client, the CqEvent is passed to the onEvent method of all CqListeners defined for the CQ.

Logical Architecture and Data Flow

Clients can execute any number of CQs, with each CQ given any number of listeners. This figure shows the logical architecture of continuous querying.
The next figure shows the typical CQ data flow when entries are updated in the server cache. A description of the data flow follows, along with a description of CQ state and life cycle.

1. Entry events come to the server's cache from any source: the server or its peers, distribution from remote sites, or updates from a client.
2. For each event, the server’s CQ executor framework checks for a match with the CQs it has running.
3. If the old or new entry value satisfies a CQ query, a CQ event is sent to the CQ's listeners on the client side. Each listener for the CQ gets the event. In the preceding figure:

- Both new and old prices for entry X satisfy the CQ query, so that event is sent indicating an update to the query results.
- The old price for entry Y satisfied the query, so it was part of the query results. The invalidation of entry Y means that it does not satisfy the query. Because of this, the event is sent indicating that it is destroyed in the query results.
- The price for the newly created entry Z does not satisfy the query, so no event is sent.

The region operations do not translate directly to specific query operations, and the query operations do not specifically describe the region events. Instead, each query operation describes how its corresponding region event affects the query results. For more information, see CqEvent Object.

State and Life Cycle
A CQ has three possible states that can be accessed from the client by calling CqQuery.getState.

- STOPPED. The CQ has been created but not yet executed, or it has been explicitly stopped from executing. The stopped CQ uses system resources. You start or restart the CQ by calling the execute method on CqQuery.
- RUNNING. The CQ is being executed on the server for all events in the region referenced by the query. Results are sent to all client listeners associated with the CqQuery.
- CLOSED. The CQ is closed and is not using system resources. Invoking an execute or stop method on closed CqQuery throws an exception.

Typical CQ life cycle
1. The client creates the CQ. This sets up everything for running the query and provides the client with a CqQuery object, but does not execute the CQ. At this point, the query is in a STOPPED state, ready to be closed or run.
2. The client runs the CQ with an API call to one of the CqQuery execute* methods. This puts the query into a RUNNING state on the client and on the server.
3. The CQ is closed by a client call to CqQuery.close. This de-allocates all resources in use for the CQ on the client and server. At this point, the cycle could begin again with the creation of a new CqQuery instance.

Implementing a Continuous Query

You can specify CQs for any client region.

Here are the high-level steps for implementing a continuous query, with links to more detailed information in this chapter.

1. Make sure your system is configured properly to run CQs. See Configuring for Continuous Querying.
2. Decide what data to track on the server and write your queries. Use your criteria for tracking data changes to write your CQ queries. See Writing the Continuous Query.
3. Determine how to handle the CQ events on the client and write your listeners.

Each CQ can have any number of listeners. In addition to your core CQ listeners, you might have listeners that you use for all CQs to track statistics or other general information. See Writing the CQ Listener.
4. Write the client code to put the queries and their listeners into named CQ queries and execute the queries. Make sure you close the queries if your client no longer needs them and when the client exits. See Running the Continuous Query Code.
5. Write any code you need for monitoring and managing your CQ query. See Managing Continuous Queries.
6. Run your code, monitor and tune as needed.
Configuring Your System for Continuous Querying

The continuous query (CQ) functionality requires standard client/server distributed system and cache configuration settings.

- The client region must use a pool with subscription-enabled set to true.
- If you want your CQs to be highly available, configure your servers for high availability as described in Configuring Highly Available Servers in the Pivotal GemFire User's Guide. When your servers are highly available, CQs are registered on primary and secondary servers, and server failover is performed without any interruption to CQ messaging. CQ events messaging uses the same queues used for server-to-client messaging.

  Note: When CQ is used with high availability, the overhead for CQs is higher than for the key-based interest list registration. CQs are executed on the primary and all secondary servers, so they require more overall server processing.

- To obtain a list of all durable CQs registered on the server, use the QueryService.getAllDurableCqsFromServer API.
- If you want your CQs to be durable, configure your native clients for durable messaging. When your clients are durable, you can create durable CQs whose events are maintained during client disconnects and replayed for the client when it reconnects. The process and data flow particular to durable CQs is described in Durable Client Messaging.
Writing the Continuous Query

Each CQ uses a query and any number of listeners. The query filters the events on the server and the listener handles the events that make it through the query filter. With the query and the listener in hand, you can create and execute your query through the API.

This is the basic syntax for the CQ query:

```
SELECT * FROM /fullRegionPath [iterator] [WHERE clause]
```

The CQ query must satisfy the standard Pivotal GemFire native client querying specifications described in Remote Querying. It also must satisfy these restrictions:

- The `FROM` clause must contain only a single region specification, with optional iterator variable.
- The query must be a `SELECT` expression only, preceded by zero or more `IMPORT` statements. This means the query cannot be a statement like `/tradeOrder.name` or `(SELECT * from /tradeOrder).size`.
- The CQ query cannot use:
  - Cross region joins
  - Drill-downs into nested collections
  - `DISTINCT`
  - Projections
  - Bind parameters

Queries not meeting these constraints generate an `UnsupportedOperationException` from the `QueryService.newCq` method.
Writing the CQ Listener or CQ Status Listener

The following C++ and C# .NET examples show how you might program a simple CqListener or CqStatusListener to update a display screen based on the CQ events it receives.

The listener retrieves the queryOperation and entry key and value from the CqEvent, then updates the screen according to the operation type provided in queryOperation.

CQ events do not change your client cache. They are provided as an event service only. This allows you to have any collection of CQs without storing large amounts of data in your regions. If you need to persist information from CQ events, program your listener to store the information where it makes the most sense for your application.

Be very careful if you choose to update your cache from your CqListener. If your listener updates the region that is queried in its own CQ, the update may be forwarded to the server. If the update on the server satisfies the same CQ, it may be returned to the same listener that did the update, which could put your application into an infinite loop. This same scenario could be played out with multiple regions and multiple CQs if the listeners are programmed to update each other's regions.

CqListener Implementation (C++)

```cpp
// CqListener class
class TradeEventListener : public CqListener {
  public:
    void onEvent(const CqEvent& cqEvent) {
      // Operation associated with the query op
      CqOperation::CqOperationType queryOperation = cqEvent.getQueryOperation();
      // key and new value from the event
      CacheableKeyPtr key = cqEvent.getKey();
      TradeOrderPtr tradeOrder = dynCast<TradeOrderPtr>(cqEvent.getNewValue());
      if (queryOperation == CqOperation::OP_TYPE_UPDATE) {
        // update data on the screen for the trade order
        ...
      } else if (queryOperation == CqOperation::OP_TYPE_CREATE) {
        // add the trade order to the screen
        ...
      } else if (queryOperation == CqOperation::OP_TYPE_DESTROY) {
        // remove the trade order from the screen
        ...
      }
    }
    void onError(const CqEvent& cqEvent) {
      // handle the error
    }
    void close() {
      // close the output screen for the trades
      ...
    }
  }
}
```

CqListener Implementation (C# .NET)

```csharp
// CqListener class
public class TradeEventListener : ICqListener {
  public void onEvent(CqEvent cqEvent) {
    // Operation associated with the query op
    CqOperationType queryOperation = cqEvent.getQueryOperation();
    // key and new value from the event
    ...
  }
}
```
Writing a CqStatusListener

If you need your CQs to detect whether they are connected to any of the servers that host its subscription queues, implement a CqStatusListener instead of a CqListener.

CqStatusListener extends the current CqListener, allowing a client to detect when a CQ is connected and/or disconnected from the server(s). The onCqConnected() method will be invoked when the CQ is connected, and when the CQ has been reconnected after being disconnected. The onCqDisconnected() method will be invoked when the CQ is no longer connected to any servers.

Taking the examples from above, we can instead implement a CqStatusListener.

When you install the CqStatusListener, your listener will be able to detect its connection status to the servers that it is querying.

CqStatusListener Implementation (C++)

class MyCqStatusListener : public CqStatusListener {
    uint8_t m_id;
    uint32_t m_numInserts;
    uint32_t m_numUpdates;
    uint32_t m_numDeletes;
    uint32_t m_numEvents;
    uint32_t m_cqsConnectedCount;
    uint32_t m_cqsDisconnectedCount;

    public:
    uint8_t getIId() {
        return m_id;
    }
    uint32_t getNumInserts() {
        return m_numInserts;
    }
    uint32_t getNumUpdates() {
        return m_numUpdates;
    }
    
```cpp
ICacheableKey key = cqEvent.getKey();
CacheableString keyStr = key as CacheableString;
IGFSerializable val = cqEvent.getNewValue();
TradeOrder tradeOrder = val as TradeOrder;
if (queryOperation==CqOperationType.OP_TYPE_UPDATE) {
    // update data on the screen for the trade order
    // . . .
} else if (queryOperation== CqOperationType.OP_TYPE_CREATE) {
    // add the trade order to the screen
    // . . .
} else if (queryOperation== CqOperationType.OP_TYPE_DESTROY) {
    // remove the trade order from the screen
    // . . .
}
```
```c
uint32_t getNumDeletes()
{
    return m_numDeletes;
}
uint32_t getNumEvents()
{
    return m_numEvents;
}
uint32_t getCqsConnectedCount()
{
    return m_cqsConnectedCount;
}
uint32_t getCqsDisconnectedCount()
{
    return m_cqsDisconnectedCount;
}
MyCqStatusListener(uint8_t id):
    m_id(id),
    m_numInserts(0),
    m_numUpdates(0),
    m_numDeletes(0),
    m_numEvents(0),
    m_cqsDisconnectedCount(0),
    m_cqsConnectedCount(0)
{
}
inline void updateCount(const CqEvent& cqEvent)
{
    m_numEvents++;  
    switch (cqEvent.getQueryOperation())
    {
    case CqOperation::OP_TYPE_CREATE: {  
        m_numInserts++;  
        break;
    }
    case CqOperation::OP_TYPE_UPDATE: {  
        m_numUpdates++;  
        break;
    }
    case CqOperation::OP_TYPE_DESTROY: {  
        m_numDeletes++;  
        break;
    }
    default:  
        break;
    }

    void onEvent(const CqEvent& cqe){
        updateCount(cqe);
    }
    void onError(const CqEvent& cqe){
        updateCount(cqe);
    }
    void close(){
    }
    void onCqDisconnected() {
        //This is called when the cq loses connection with all servers.
        m_cqsDisconnectedCount++;
    }
    void onCqConnected() {
        //This is called when the cq establishes a connection with a server.
        m_cqsConnectedCount++;
    }
    void clear() {
        m_numInserts = 0;
        m_numUpdates = 0;
        m_numDeletes = 0;
        m_numEvents = 0;
        m_cqsDisconnectedCount = 0;
        m_cqsConnectedCount = 0;
    }
```
CQStatusListener Implementation (C# .NET)

```csharp
public class MyCqStatusListener<TKey, TResult> : ICqStatusListener<TKey, TResult>
{
    #region Private members
    private bool m_failedOver = false;
    private UInt32 m_eventCountBefore = 0;
    private UInt32 m_errorCountBefore = 0;
    private UInt32 m_eventCountAfter = 0;
    private UInt32 m_errorCountAfter = 0;
    private UInt32 m_CqConnectedCount = 0;
    private UInt32 m_CqDisConnectedCount = 0;
    #endregion

    #region Public accessors
    public MyCqStatusListener(int id)
    {
    }
    public void failedOver()
    {
        m_failedOver = true;
    }
    public UInt32 getEventCountBefore()
    {
        return m_eventCountBefore;
    }
    public UInt32 getErrorCountBefore()
    {
        return m_errorCountBefore;
    }
    public UInt32 getEventCountAfter()
    {
        return m_eventCountAfter;
    }
    public UInt32 getErrorCountAfter()
    {
        return m_errorCountAfter;
    }
    public UInt32 getCqConnectedCount()
    {
        return m_CqConnectedCount;
    }
    public UInt32 getCqDisConnectedCount()
    {
        return m_CqDisConnectedCount;
    }
    #endregion

    public virtual void OnEvent(CqEvent<TKey, TResult> ev)
    {
        if (m_failedOver == true)
        {
            m_eventCountAfter++;
        }
        else
        {
            m_eventCountBefore++;
        }
    }
    public virtual void OnError(CqEvent<TKey, TResult> ev)
    {
        if (m_failedOver == true)
        {
            m_errorCountAfter++;
        }
        else
        {
            m_errorCountBefore++;
        }
    }
    public virtual void Close()
    {
    }
```
public virtual void OnCqConnected()
{
    m_CqConnectedCount++;
}
public virtual void OnCqDisconnected()
{
    m_CqDisConnectedCount++;
}
public virtual void Clear()
{
    m_eventCountBefore = 0;
    m_errorCountBefore = 0;
    m_eventCountAfter = 0;
    m_errorCountAfter = 0;
    m_CqConnectedCount = 0;
    m_CqDisConnectedCount = 0;
}
CqEvent Object

The CqEvent object contains information about the CQ event.

- Entry key and new value.
- Base operation that triggered the CQ event in the server.
- CqQuery object associated with this CQ event.
- Query operation associated with this CQ event. This operation describes the change affected to the query results by the cache event. Possible values are:
  - CREATE, which corresponds to the standard database INSERT operation.
  - UPDATE
  - DESTROY, which corresponds to the standard database DELETE operation.

This table describes the query operation based on whether the old and new entry values in the region entry event satisfy the query criteria.

**Table 23: Query Operation Based on Old and New Entry Values**

<table>
<thead>
<tr>
<th>Old Entry Value</th>
<th>New Entry Value</th>
<th>Query Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>No value or value does not satisfy the query criteria.</td>
<td>No value (operation is invalidate or destroy) or value does not satisfy the query.</td>
<td>N/A - no event</td>
</tr>
<tr>
<td></td>
<td>Value satisfies the query.</td>
<td>create</td>
</tr>
<tr>
<td>Value satisfies the query</td>
<td>No value (operation is invalidate or destroy) or value does not satisfy the query.</td>
<td>destroy</td>
</tr>
<tr>
<td></td>
<td>Value satisfies the query.</td>
<td>update</td>
</tr>
</tbody>
</table>

You can use the query operation to decide what to do with the CqEvent in your listeners. For example, a CqListener that displays query results on screen might stop displaying the entry, start displaying the entry, or update the entry display depending on the query operation.
Running the Continuous Query Code

Create your CQ from an instance of the QueryService. Once created, the CQ is maintained primarily through the CqQuery interface. The following two C++ and C# examples show the basic calls in the CQ life cycle.

CQ Creation, Execution, and Close (C++)

```
// Get cache and qrySvcPtr - refs to local cache and QueryService
// Create client /tradeOrder region configured to talk to the server
// Create CqAttribute using CqAttributeFactory
CqAttributesFactory cqf;
// Create a listener and add it to the CQ attributes
// callback defined below
CqListenerPtr tradeEventListener (new TradeEventListener());
QueryServicePtr qrySvcPtr = cachePtr->getQueryService();
cqf.addCqListener(tradeEventListener);
CqAttributesPtr cqa = cqf.create();
// Name of the CQ and its query
char* cqName = "priceTracker";
char* queryStr = "SELECT * FROM /tradeOrder t where t.price > 100.00";
// Create the CqQuery
CqQueryPtr priceTracker = qrySvcPtr->newCq(cqName, queryStr, cqa); try {
// Execute CQ
priceTracker->execute();
} catch (Exception& ex) {
    ex.printStackTrace();
}
// Now the CQ is running on the server, sending CqEvents to the listener
...
// End of life for the CQ - clear up resources by closing
priceTracker->close();
```

CQ Creation, Execution, and Close (C# .NET)

```
// Get cache and queryService - refs to local cache and QueryService
// Create client /tradeOrder region configured to talk to the server
// Create CqAttribute using CqAttributeFactory
CqAttributesFactory cqf = new CqAttributesFactory();
// Create a listener and add it to the CQ attributes
// callback defined below
ICqListener tradeEventListener = new TradeEventListener();
cqf.addCqListener(tradeEventListener);
CqAttributes cqa = cqf.create();
// Name of the CQ and its query
String cqName = "priceTracker";
String queryStr = "SELECT * FROM /tradeOrder t where t.price > 100.00 ";
QueryService queryService = cache.GetQueryService();
// Create the CqQuery
CqQuery priceTracker = queryService.newCq(cqName, queryStr, cqa, true); try {
// Execute CQ
priceTracker.execute();
} catch (Exception ex) {
    // handle exception
}
// Now the CQ is running on the server, sending CqEvents to the listener
// ...
// End of life for the CQ - clear up resources by closing
```
priceTracker.close();
CQ Execution Options

CQ execution can be done with or without an initial result set by calling CqQuery.Execute or CqQuery.ExecuteWithInitialResults. The initial SelectResults returned from ExecuteWithInitialResults is the same as the one you would get if you ran the query ad hoc by calling QueryService.NewQuery.Execute on the server cache, but with the key added.

Individual CQs are executed using CqQuery.execute* methods. You can also execute all CQs for the client or for a region through the client QueryService. CQs that are running can be stopped or closed.

If you are managing a data set from the CQ results, you can initialize the set by iterating over the result set and then updating it from your listeners as events arrive. For example, you might populate a new screen with initial results and then update the screen from a listener.

Just as with the standalone query, the initial results represents a possibly moving snapshot of the cache. If there are updates to the server region while the result set is being created, the result set and the subsequent event-by-event CQ query execution might miss some events.
When an Error Occurs in a Running CQ

When an error occurs in CQ execution on the server, specific information on the error itself is stored in the server's log file. An exception is passed to the client, and the client throws an exception.

The server log will contain an error message indicating an error while processing CQ, like this:

```
[error 2007/12/18 12:03:18.903 PST gemfire1 <RMI TCP Connection(2)-10.80.10.91> tid=0x18] Error while processing CQ on the event, key :
key-1, CqName :testCQEvents_0, ClientId :
identity(carlos(3249):52623/35391,connection=1,durableAttributes=null)
Error :No public attribute named 'ID' was found in class java.lang.Object
```

Errors in CQ execution are usually caused by data errors, such as invalid object types that are stored in the server region. In this case, the query is trying to read into an object of type Portfolio for an entry where an empty object has been stored. You can avoid these types of errors by placing constraints on the region entries, or by otherwise controlling the types of objects stored in your server regions.

Managing Continuous Queries

This section discusses how to access and manage your CQs from your client. The calls discussed here are all executed specifically for the calling client. A client cannot access or modify the CQs belonging to another client.

For detailed method usage, see the online API documentation.

Accessing CQs and CQ Statistics

You can use the `QueryService.getCq*` methods to access a single named CQ, an array of all CQs registered, and an array of all CQs registered in the client. You can use the `CqEvent.getCq` method to access the CQ used to produce a `CqEvent`.

CQ runtime statistics are available for the client through the `CqServiceStatistics` and `CqStatistics` interfaces described under "CQ API and Main Features." You can get information on the events generated by a specific CQ from the `CqStatistics` object returned by `CqQuery.GetStatistics`. You can get higher-level information about the CQs the client has registered, running, and so on, from the `CqServiceStatistics` object returned by `QueryService.GetCqStatistics`.

For both the client and server, you can access these statistics by loading the statistics archive file into VSD. The VSD (Visual Statistics Display) tool can be acquired as part of the GemFire main product download.

Client statistics are for the single client only. The server's pertain to all clients with CQs on this server.

Modifying CQ Attributes

You can modify the attributes for an existing CQ using the methods provided by `CqQuery.GetCqAttributesMutator`. The attributes consist of a list of listeners.
Stopping and Closing CQs

You stop individual CQs with the `CqQuery.stop` method. You can stop all CQs for the client through the `QueryService`. Stopped CQs are in the same state as new CQs that have not yet been executed. You can close or execute a stopped CQ.

You close individual CQs with the `CqQuery.close` method. You can also close all CQs for the client through the `QueryService`. Closed CQs cannot be executed. CQs are also closed in the following cases:

- The client closes its cache after closing all of its CQs--Closing the client cache closes the `QueryService` and all associated CQs on the client and server.
- The client disconnects from its server--This might be because of a network outage or some other failure. When a client disconnects, all CQs created by the client are removed from the server and put into a `CLOSED` state on the client.
- The server region is destroyed--When a server region is destroyed, all associated CQs are also cleaned up on the server and the region destroy event is sent to the client. On the client, the `CqListener.Close` method is called for all CQs on the region.

Getting All Durable CQs Registered with the Server

To obtain a list of all durable CQs registered on the server, use the `QueryService.getAllDurableCqsFromServer` API.

CQ API and Main Features

This chapter documents the primary native client API for CQ management.

The Pivotal GemFire native client API allows your clients to create and manage CQs. (The server side does not have an API.) Continuous querying provides native client query syntax, events-based management, integration with client/server architecture, active query execution, and interest criteria based on data values. For complete information on the classes and interfaces described here, see the online API documentation.

- `Gemstone::GemFire::Cache`
- `Main Features of Continuous Querying`

Gemstone::GemFire::Cache

Only C# versions of CQ API interfaces, classes, and methods are shown here (example: `CqQuery.execute`). The code examples demonstrate both C++ and C# versions.

- `QueryService` interface. Provides methods that enable you to:
  - Create a new CQ and specify whether it is durable (available for durable clients)
  - Execute a CQ with or without an initial result
  - List all the CQs registered by the client
  - Close and stop CQs
  - Get a handle on `CqStatistics` for the client

You run `QueryService` CQ methods against the server cache. The `QueryService` can be obtained from the cache or from a pool.

- `CqQuery` interface. Provides methods for managing a continuous query after it is created through the `QueryService`. This interface is used typically to begin and end CQ execution and to retrieve other CQ-related objects such as CQ attributes, CQ statistics, and CQ state.

- `CqListener` application plug-in interface. Handles continuous query events after they occur. You program this interface.
- **CqEvent interface.** Provides all information sent from the server about the CQ event, which is passed to the CQ's Cqlistener methods.
- **CqState interface.** Provides information on the state of a CqQuery, through the getState method of the CqQuery instance.
- **CqAttributes, CqAttributesFactory, CqAttributesMutator interfaces.** Allow you to manage CQ attributes. The attributes consist of Cqlistener plug-in specifications.
- **CqStatistics, CqServiceStatistics interfaces.** Allow you to access statistics information for a single CQ and for the query service's management of CQs as a whole. For details on statistics, see Statistics API.

**Main Features of Continuous Querying**

Continuous querying in the native client has the following features:

- **Standard GemFire native client query syntax and semantics.** CQ queries are expressed in the same language used for other native client queries (see Remote Querying.).
- **Standard GemFire events-based management of CQ events.** The event handling used to process CQ events is based on the standard GemFire Enterprise event handling framework. The Cqlistener interface is similar to Application Plug-Ins and Application Callbacks.
- **Complete integration with the client/server architecture.** CQ functionality uses existing server-to-client messaging mechanisms to send events. All tuning of your server-to-client messaging also tunes the messaging of your CQ events. If your system is configured for high availability then your CQs are highly available, with seamless failover provided in case of server failure (see High Availability for Client-to-Server Communication). If your clients are durable, you can also define any of your CQs as durable (see Durable Client Messaging).
- **Interest criteria based on data values.** CQ queries are run against the region's entry values. Compare this to register interest by reviewing Registering Interest for Entries.
- **Active query execution.** Once initialized, the queries only operate on new events instead of on the entire region data set. Events that change the query result are sent to the client immediately.
Part XIV

Using Connection Pools

*Using Connection Pools* describes how connection pools achieve load balancing for the client and describes how to configure connection pools as server locators or as a list of servers.

How Client Load Balancing Works

In a distributed system, servers can be added or removed and their capacity to service new client connections may vary. The server connectivity options are specified in the connection pool configuration. The Pivotal GemFire connection pool API supports connecting to servers through server locators or directly connecting to servers.
Server Locators

Server locators continuously monitor server availability and server load information. The client is configured with a list of server locators and consults a server locator to request a connection to a server in the distributed system.

Locators provide clients with dynamic server discovery and server load balancing. They give clients connection information for the server with the least load at any given time.

Server locators provide these main features:

- **Automated discovery of servers and locators.** Adding and removing servers or locators is made easy as each client does not require a list of servers to be configured at the time of pool creation.
- **Client load rebalancing.** Server locators give clients dynamic server information and provide server load rebalancing after servers depart or join the system.
- **High availability.** When a client/server connection receives an exception, the connection is automatically failed over to another available connection in the pool. Redundancy is also provided for client subscriptions.

Alternatively, you can configure a pool statically with a list of endpoints. When the pools are statically configured, a round-robin load balancing policy is used to distribute connections across the servers.
Connection Pools

Clients contain connection pools. Each region is associated with a connection pool using a region attribute, and operations on the region use connections from the respective pools.

The server connectivity options are specified in the connection pool configuration. Each pool has a minimum and maximum number of connections.

Each cache operation that requires server connectivity obtains a connection from the pool for the server group that the operation affects, performs the operation using the connection, and returns the connection to the pool. If the pool size is less than the maximum number of connections and all connections are in use, the connection pool creates a new connection and returns it. If the pool is at the maximum number of connections, that thread blocks until a connection becomes available or a free-connection-timeout occurs. If a free-connection-timeout occurs, an AllConnectionsInUse exception is thrown.

The connection pool has a configurable timeout period that is used to expire idle connections. The idle connections are expired until the pool has the minimum number of connections. A monitoring thread expires idle connections, adds sufficient connections to bring up the count to minimum, closes connections whose lifetime has been exceeded. See the load-conditioning-interval and idle-timeout attributes of the <pool> element. A separate thread (ping) tests each connected endpoint for its status and if the endpoint is not reachable, the thread closes all connections that have been made to the endpoint. See the ping-interval attribute of the <pool> element.

When a connection receives an exception, the operation is failed over to another connection from the pool. The failover mechanism obtains the endpoint to failover to from the locator or from the specified endpoint list in the pool.

Figure 7: Logical Architecture of Client/Server Connections
Discovering Locators Dynamically

A background thread periodically queries the locator for any other locators joining the distributed system. To balance load across all the locators, locators are shuffled before pushing in the list.

However, if locator A (to which the client is connected) goes down before it discovers locator B, the locator B is never discovered even though it is alive and the client receives a NoLocatorsAvailable exception.

One connection is attached to every application thread that is local to the respective thread. This is known as a thread local connection.

In this case, to perform any cache operation the client is not required to obtain a connection from pool. Instead the thread local connection of the client is used.

A thread local connection can be released by invoking the Pool::releaseThreadLocalConnection() method. The released connection is returned to the pool. If the number of threads is larger than the number of max-connections, the client throws an AllConnectionsInUseException after the free-connection-timeout lapses, unless the Pool::releaseThreadLocalConnection() method is used judiciously.

If a connection expires or the server goes down on which the connection was established, a thread local connection is immediately replaced with a good connection obtained from the pool.

Configuring Pools

A pool can be configured as locators or as a list of servers.

You configure locator, server, and pool settings declaratively in the client's cache.xml file or programmatically through the PoolFactory method. You create an instance of PoolFactory through PoolManager.
Native Client Pool API

The GemFire native client API allows your clients to create and manage connection pools. The server side does not have an API.

This section lists the primary native client API for pool management. For complete information on the classes and interfaces described here, see the online API documentation.

**Note**: Only C# versions of Pool API interfaces, classes, and methods are shown throughout the text in this chapter (example: `Pool.GetQueryService()`). The code examples demonstrate both C++ and C# versions.

### Gemstone::GemFire::Cache

- **Pool** interface. API to retrieve pool attributes.
- **PoolFactory** interface. API to configure pool attributes.
- **PoolManager** interface. API to create a PoolFactory object and to find the pool objects.
- **AttributesFactory** class. Has a new method `setPoolname` which assigns a pool to a region. Operations performed on the configured region use connections from the pool.

  **Note**: A region can have a pool attached to it. A pool may have multiple regions attached to it.
Pool Configuration Example and Settings

Connection pools require standard client/server distributed system and cache configuration settings. You must also configure settings for the locator, server, and pool elements.

- Locator. Host and port where a server locator is listening.
- Server. Host and port where a server is listening.
- Pool. Client/server connection pool.

The example shows a declarative pool configuration. Following the example is a table that describes the attributes that can be configured.

Example: Declarative Pool Configuration

This example shows a declarative pool configuration.

**Note:** You create an instance of `PoolFactory` through `PoolManager`.

```xml
<pool free-connection-timeout="12345" idle-timeout="5555"
load-conditioning-interval="23456"
max-connections="7" min-connections="3"
name="test_pool_1" ping-interval="12345"
read-timeout="23456" retry-attempts="3" server-group="ServerGroup1"
socket-buffer-size="32768" statistic-interval="10123"
subscription-ack-interval="567" subscription-enabled="true"
subscription-message-tracking-timeout="900123"
subscription-redundancy="0" thread-local-connections="true">
  <locator host="localhost" port="34756"/>
</pool>
```

Pool Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>free-connection-timeout</td>
<td>Number of milliseconds (ms) that the client waits for a free connection if max-connections limit is configured and all connections are in use.</td>
<td>10000 ms</td>
</tr>
<tr>
<td>idle-timeout</td>
<td>Number of milliseconds to wait for a connection to become idle for load balancing</td>
<td>5000 ms</td>
</tr>
<tr>
<td>load-conditioning-interval</td>
<td>Interval in which the pool checks to see if a connection to a specific server should be moved to a different server to improve the load balance.</td>
<td>300000 ms (5 minutes)</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>max-connections</td>
<td>Maximum number of connections that the pool can create. If all connections are in use, an operation requiring a client-to server-connection is blocked until a connection is available or the free-connection-timeout is reached. If set to -1, there is no maximum. The setting must indicate a cap greater than min-connections. <strong>Note:</strong> If you use this setting to cap your pool connections, disable the pool attribute pr-single-hop-enabled. Leaving single hop enabled can increase thrashing and lower performance.</td>
<td>-1</td>
</tr>
<tr>
<td>min-connections</td>
<td>Number of connections that must be created initially.</td>
<td>5</td>
</tr>
<tr>
<td>name</td>
<td>Pool name.</td>
<td></td>
</tr>
<tr>
<td>ping-interval</td>
<td>Interval between pinging the server to show the client is alive, set in milliseconds. Pings are only sent when the ping-interval elapses between normal client messages. This must be set lower than the server’s maximum-time-between-pings.</td>
<td>10000 ms</td>
</tr>
<tr>
<td>pr-single-hop-enabled</td>
<td>Setting used for single-hop access to partitioned region data in the servers for some data operations. See PartitionResolver. See note in thread-local-connections below.</td>
<td>True</td>
</tr>
<tr>
<td>read-timeout</td>
<td>Number of milliseconds to wait for a response from a server before the connection times out.</td>
<td>10000</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>retry-attempts</td>
<td>Number of times to retry an operation after a time-out or exception for high availability. If set to -1, the pool tries every available server once until it succeeds or has tried all servers.</td>
<td>-1</td>
</tr>
<tr>
<td>server-group</td>
<td>Server group from which to select connections. If not specified, the global group of all connected servers is used.</td>
<td>empty</td>
</tr>
<tr>
<td>socket-buffer-size</td>
<td>Size of the socket buffer, in bytes, on each connection established.</td>
<td>32768</td>
</tr>
<tr>
<td>statistic-interval</td>
<td>Default frequency, in milliseconds, with which the client statistics are sent to the server. A value of -1 indicates that the statistics are not sent to the server.</td>
<td>-1</td>
</tr>
<tr>
<td>subscription-ack-interval</td>
<td>Number of milliseconds to wait before sending an acknowledgment to the server about events received from the subscriptions.</td>
<td>100</td>
</tr>
<tr>
<td>subscription-enabled</td>
<td>Whether to establish a server to client subscription.</td>
<td>False</td>
</tr>
<tr>
<td>subscription-message-tracking-timeout</td>
<td>Number of milliseconds for which messages sent from a server to a client are tracked. The tracking is done to minimize duplicate events.</td>
<td>90000</td>
</tr>
<tr>
<td>subscription-redundancy</td>
<td>Redundancy for servers that contain subscriptions established by the client. A value of -1 causes all available servers in the specified group to be made redundant.</td>
<td>0</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>thread-local-connections</td>
<td>Whether the connections must have affinity to the thread that last used them.</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> To set this to true, also set pr-single-hop-enabled to false. A true value in pr-single-hop-enabled automatically assigns a false value to thread-local-connections...</td>
<td></td>
</tr>
<tr>
<td>update-locator-list-interval</td>
<td>An integer number of milliseconds defining the interval between locator list updates. If the value is less than or equal to 0, the update will be disabled.</td>
<td>5000</td>
</tr>
</tbody>
</table>
Subscription Properties

Each connection pool has a single subscription connection that can be to any server that matches the requirements of the connection pool.

When a client registers interest for a region, if the connection pool does not already have a subscription channel, the connection pool sends a message to the server locator, and the server locator chooses servers to host the queue and return those server names to the client. The client then contacts the chosen servers and asks them to create the queue.

The client maintains at least one connection with each server hosting a queue. If the server does not detect any connections from a non-durable client, it drops the client queue and closes all artifacts for the client. For information about durable client subscriptions, see Durable Client Messaging.

Requesting a Subscription Region Queue

The client-to-server locator request is a short lived TCP request. The client sends a message with:

- The client ID.
- (Optional) target server group.
- Number of redundant copies.
- Servers to exclude from the results. This list is used if the client cannot connect to a server and needs to request a new one.

The server locator responds with a list of servers. The client is responsible for contacting the primary and secondaries and asking them to host the queue.

For durable subscriptions, the server locator must be able to locate the servers that host the queues for the durable client. When a durable client sends a request, the server locator queries all the available servers to see if they are hosting the subscription region queue for the durable client. If the server is located, the client is connected to the server hosting the subscription region queue.
Running the Connection Pool Code

Examples demonstrate a simple procedure to create a pool factory and then create a pool instance in C++ and C#. They also help you to execute a query.

The examples create a pool with locators. Ensure that you create a pool with locators or endpoints, but not both. The first example demonstrates creating a pool by adding locators. The second example demonstrates creating a pool by adding servers. For more information, see the example in the QuickStart Guide.

**Connection Pool Creation and Execution Using C++**

```cpp
PropertiesPtr prptr = Properties::create();
systemPtr = CacheFactory::createCacheFactory(prptr);

  cachePtr = systemPtr->create();
  PoolFactoryPtr poolFacPtr = PoolManager::createFactory();
  //to create pool add either endpoints or add locators or servers
  //pool with endpoint, adding to pool factory
  //poolFacPtr->addServer("localhost", 12345 /*port number*/);
  //pool with locator, adding to pool factory
  poolFacPtr->addLocator("localhost", 34756 /*port number*/);
  PoolPtr pptr = NULLPTR;
  if ((PoolManager::find("examplePool")) == NULLPTR) {
    // Pool does not exist with the same name.
    pptr = poolFacPtr->create("examplePool");
  }  

  RegionFactoryPtr regionFactory =
    cachePtr->createRegionFactory(CACHING_PROXY);
  regionPtr = regionFactory
    ->setPoolName("examplePool")
    ->create("regionName");
  QueryServicePtr qs = cachePtr->getQueryService("examplePool");
```

**Connection Pool Creation and Execution Using (C#.NET)**

```csharp
Properties prop = Properties.Create();
CacheFactory cacheFactory = CacheFactory.CreateCacheFactory(prop);
Cache cache = cacheFactory.Create();

PoolFactory poolFact = PoolManager.CreateFactory();
//to create pool add either endpoints or add locators
//pool with endpoint, adding to pool factory.
poolFact.AddServer("localhost", 40404 /*port number*/);
//pool with locator, adding to pool factory
//poolFact.AddLocator("hostname", 15000 /*port number*/);
Pool pool = null;
if (PoolManager.Find("poolName") == null) {
  pool = poolFact.Create("poolName");
}

int loadConditInterval = pool.LoadConditioningInterval;
RegionFactory regionFactory = cache.CreateRegionFactory(RegionShortcut.CACHING_PROXY);
IRegion<string, string> region =
  regionFactory.SetPoolName(poolName).Create<string,
  string>(regionName);
```
Part XV

Transactions

Transactions describes how transactions work on the client side. It provides examples for running, suspending, and resuming transactions.

Note: Pivotal GemFire client transactions run on the server tier. The client uses a server delegate that runs the transaction as it would a local transaction. Thus the key to running client transactions lies in making sure the server is properly configured and programmed. For complete information about GemFire transactions in the Java server, see the Transactions chapter in the GemFire User’s Guide. The User’s Guide provides detailed information including server data requirements, interactions of GemFire transactions with other operations running on the server tier, server-side application plug-ins with transactions, and querying with transactions.

How Native Client Transactions Work

The syntax for writing client transactions is the same as with server or peer transactions, but when a client performs a transaction, the transaction is delegated to a server that brokers the transaction.

- Role of Server Delegates in Transactions
- Native Client Transaction APIs

Role of Server Delegates in Transactions
The Pivotal GemFire native client can run transactions on the Java cache server, using a server delegate to actually run the transaction code.

For information on transaction requirements and activities on the server side, see the Transactions chapter in the GemFire User’s Guide.

Note: The client cache blocks until the transaction is successfully committed. However, the client cache block is removed if the transaction is suspended.

Depending on where the data resides, the server transaction delegate may or not be the same member that hosts the transaction. This is the same as for transactions run by the servers, but for server-run transactions, there is no delegate. There is just the member that is directly running its own transaction code.

In this figure, the application code on the client makes changes to data entries Y and Z within a transaction. The server delegate that performs the transaction, M1, does not host the primary copy of the data being modified. The transaction takes place on server M2, where the data resides.
To maintain cache consistency, the local client cache is not accessible during a transaction as it may reflect information inconsistent with the transaction in progress. When the transaction completes, the local cache is accessible again.

In addition to the failure conditions common to all transactions, client transactions can also fail if the transaction delegate fails. If the delegate performing the transaction fails, the transaction code throws a `TransactionException`.

**Figure 8: Transaction Run From a Client**

### Native Client Transaction APIs

The GemFire API for distributed transactions has the familiar relational database methods, `begin`, `commit`, and `rollback`. There are also APIs available to suspend and resume transactions.

The .NET classes for executing transactions are:

- `GemStone::GemFire::Cache::Generic`
  - `CacheTransactionManager`
  - `TransactionId`

---

**Running a Native Client Transaction**

Before you can run a native client transaction, you must configure your clients and servers; define your server regions for your transactions; and define your client regions.

For information on server and server region configuration, see the *GemFire User's Guide*.

1. Retrieve the GemFire cache transaction manager.
transactions

# Transactions

## C++ example

```cpp
CacheTransactionManagerPtr txManager = cache->getCacheTransactionManager();
```

## C# .NET example

```csharp
CacheTransactionManager txManager = cache.CacheTransactionManager;
```

2. Run your transaction. (Detailed steps follow the examples.)

## C++ example

```cpp
TransactionIdPtr tid;
    txManager->begin();
    // ..do work
    tid = txManager->suspend();
    // following code can be run from another
    // thread that has access to tid
    try {
        txManager->resume(tid);
        // ..do work
        tid = txManager->commit();
        catch (const CommitConflictException& e)
    }
```

## C# .NET example

```csharp
TransactionId tid;
    txManager.Begin();
    // ..do work
    tid = txManager.Suspend();
    // following code can be run from another
    // thread that has access to tid
    try {
        txManager.Resume(tid);
        // ..do work
        txManager.Commit();
        catch (CommitConflictException e)
    }
```

- Start each transaction with a `begin` operation.
- If your transaction runs on server regions that are a mix of partitioned and replicated regions, perform your first transaction operation on a partitioned region. This sets the server data host for the entire transaction. If you are using PR single-hop, single-hop will be applied as usual to this first operation.
- Run the GemFire operations and other operations that you want included in the transaction.
- End each transaction with a `commit` or a `rollback`.

**Note:** Do not leave any transaction in an uncommitted and unrolled back state unless you have suspended the transaction. Transactions that have not been explicitly suspended do not time out, so will remain in the system for the life of your application.

3. Review all of your client code for compatibility with transactions.

When you commit a transaction, while the commit is taking place, the changes are visible in the cache. This is also known as transition commits. This provides better performance than locking everything to do the transaction updates, but it means that another process accessing data used in the transaction might get some data in the pre-transaction state and some in the post-transaction state.

For example, key 1 and 2 are written to in a transaction so both of their values change from A to B. In another thread, it is possible to read key 1 with value B and key 2 with value A, while the transaction is...
being committed. This can happen because of how GemFire performs reads. This choice sacrifices atomic
visibility in favor of performance. Reads don't block writes. Writes don't block reads.

Because the client cache waits during transaction execution, and client regions are not distributed, the only
activities that interact with a client transaction are those that occur on the server.

**Suspending and Resuming Transactions**

The ability to suspend and resume transactions is useful when a thread must perform operations that
should not be part of the transaction before the transaction can complete.

When a transaction is suspended, it loses the transactional view of the cache. None of the previous
operations (before calling suspend) are visible to the thread. Subsequently any operations that are
performed by the thread do not participate in the suspended transaction.

When a transaction is resumed, the resuming thread assumes the transactional view. A transaction that is
suspending on a member must be resumed on the same member. Before resuming a transaction, you may
want to check if the transaction exists on the member and whether it is suspended. You may optionally use
the `tryResume` method.

If the member with the primary copy of the data crashes, the transactional view that applied to that data is
lost. The secondary member for the data cannot resume transactions suspended on the crashed member.
You need to take remedial steps to retry the transaction on a new primary copy of the data.

If a suspended transaction is not touched for a period of time, Pivotal GemFire cleans it up automatically.
By default, the timeout for a suspended transaction is 30 minutes and can be configured by using the
`suspended-tx-timeout` property of the `gfcpp.properties` file. The suspended transaction timeout value
is specified in milliseconds.

See *Running a Native Client Transaction* for code examples of how to suspend and resume a transaction.
Part XVI

Function Execution

*Function Execution* describes how you can execute application functions to achieve linear scalability. It explains how function execution works and lists specific use cases.

**Note:** Function execution can be used only along with the pool functionality. For more information about the pool API, see *Using Connection Pools*. Only C++ versions of Function Execution API interfaces, classes, and methods (like `FunctionService::onRegion`) are shown in text. The code examples show C++ and C#.

Understanding Data-Aware Function Routing

Achieving linear scalability is predicated upon being able to horizontally partition the application data such that concurrent operations by distributed applications can be done independently across partitions.

In other words, if the application requirements for transactions can be restricted to a single partition, and all data required for the transaction can be colocated to a single server member or a small subset of server members, then true parallelism can be achieved by vectoring the concurrent accessors to the ever-growing number of partitions.

Most scalable enterprise applications grow in data volume, where the number of data items managed rather than the size of individual items grows over time. If the above logic holds (especially true for OLTP class applications), then we can derive sizable benefits by routing the data-dependent application code to the fabric member hosting the data. This routing of application code to the data of interest is called data-aware function routing, or behavior routing.

How Functions Execute

This chapter discusses the basic function execution process, how highly available functions execute after a failure, and the execution scenarios for data-dependent and data-independent functions.

**How Functions Execute**

1. The calling client application runs the `execute` method on the Function Execution object. The object must be already registered on the servers.
2. The execution is sent to all servers where it needs to run. The locations are determined by the `FunctionService on*` method calls, region configuration, and any filters.
3. If the function has results, the result is returned to the `execute` method call in a `ResultCollector` object.
4. The client collects results using the result collector `getResult`.
How Highly Available Functions Execute after a Failure

If a failure occurs in function execution, the error is returned to the calling application. You can code for high availability for `onRegion` functions that return a result, so the function is automatically retried. For information on setting this up on the server side, see *Executing a Function in Pivotal GemFire* in the GemFire User's Guide. To use a highly available function, the client must call the results collector `getResult` method. When an execution error occurs or a member crashes while executing, the system does the following:

1. Waits for all calls to return.
2. Sets a boolean indicating a reexecution is being done.
3. Calls the result collector's clearResults method.
4. Executes the function.

The system retries the execution up to the number specified in the server pool's `retryAttempts` setting. If the function continues to fail, the final exception is returned to the `getResult` method.

Data-Independent Function Execution

The figure shows the sequence of events for a data-independent function executed against all available servers.

![Data-Independent Function Invoked from a Client](image)

Data-Dependent Function Execution

The figure shows a data-dependent function run by a client. The specified region is connected to the server system, so the function automatically goes there to run against all servers holding data for the region.
Figure 10: Data-Dependent Function Invoked From a Client

This shows the same data-dependent function with the added specification of a set of keys on which to run. Servers that don't hold any of the keys are left out of the function execution.
Figure 11: Data-Dependent Function with Filter Invoked from a Client

This scenario demonstrates the steps in a call to a highly available function. The call fails the first time on one of the participating servers and is successfully run a second time on all servers.
Figure 12: Highly Available Data-Dependent Function with Failure on First Executions
Executing Functions in GemFire

Using the Pivotal GemFire function execution service, you can execute application functions on a single server member, in parallel on a subset of server members, or in parallel on all server members of a distributed system.

In these procedures, it is assumed that you have defined your client and server regions, and that you have coded and configured your servers to run your functions. In the Function Execution chapter of the GemFire User’s Guide, see the function execution information for the server side.
Running the Function

In this section you create an Execution object and use its methods to define and run the function. To run a function with high availability, you call `getResult` from the results collector returned from the `execute` method.

- **Configuring and Running a Function**
- **Running a Function on a Region (C++)**
- **Running a Function on a Server Pool (C++)**
- **Running a Function on a Region (C# .NET)**
- **Running a Function on a Server Pool (C# .NET)**

### Configuring and Running a Function

You specify the members that run the function and, optionally, the data set over which the functions run.

- **Servers.** Execute the function in a single server or a set of servers, specified by the server pool. To specify data sets for this type of function, pass arguments in to the function.
- **Data set.** Specify a region and possibly a set of keys on which to run.

In every client where you want to execute the function and process the results:

1. Use one of the `FunctionService on*` methods to create an `Execution` object. The `on*` methods, `onRegion`, `onServer` and `onServers`, define the highest level where the function is run. If you use `onRegion` you can further narrow your run scope by setting key filters. The function run using `onRegion` is a data dependent function – the others are data-independent functions.

   You can run a data dependent function against custom partitioned and colocated partitioned regions. The steps for setting up the regions and functions in the server are provided in the GemFire User’s Guide. From the client, just provide the appropriate key sets to the function call.

2. Use the `Execution` object as needed for additional function configuration. You can:
   - Provide a set of data keys to `withFilter` to narrow the execution scope. This works only for `onRegion Execution` objects.
   - Provide function arguments to `withArgs`.
   - Define a custom `ResultCollector` to `withCollector`. See Programming to Get Function Results.

3. Call the `Execution` object `execute` method to run the function.

4. To run a function with high availability, call `getResult` from the results collector returned from `execute`. Calling a highly available function without using `getResult` disables the high availability functionality.

#### Running a Function on a Region (C++)

```cpp
regPtr0 = initRegion();
ExecutionPtr exc = FunctionService::onRegion(regPtr0);
CacheableVectorPtr routingObj = CacheableVector::create();
char buf[128];
bool getResult = true;
sprintf(buf, "VALUE--%d", 10);
CacheablePtr value(CacheableString::create(buf));
regPtr0->put(key, value);
sprintf(buf, "KEY--%d", 100);
CacheableKeyPtr key = CacheableKey::create(buf);
regPtr0->put(key, value);
CacheableKeyPtr key1 = CacheableKey::create(buf);
routingObj->push_back(key1);
```
CacheablePtr args = routingObj;
CacheableVectorPtr executeFunctionResult = exc->withFilter(routingObj)->
    withArgs(args)->execute(func, getResult)->getResult();

Running a Function on a Server Pool (C++)

pptr = PoolManager::find(poolName);
ExecutionPtr exc = FunctionService::onServer(cache);
CacheableVectorPtr routingObj = CacheableVector::create();
char buf[128];
bool getResult = true;
sprintf(buf, "VALUE--%d", 10);
CacheablePtr value(CacheableString::create(buf));

sprintf(buf, "KEY--%d", 100);
CacheableKeyPtr key = CacheableKey::create(buf);
regPtr0->put(key, value);

sprintf(buf, "KEY--%d", 100);
CacheableKeyPtr key1 = CacheableKey::create(buf);
routingObj->push_back(key1);

CacheablePtr args = routingObj;
CacheableVectorPtr executeFunctionResult =
    exc->withArgs(args)->execute(func, getResult)->getResult();

Running a Function on a Region (C# .NET)

IRegion<string, string> fregion =
    regionFactory.Create<string, string>("exampleRegion");
for (int i = 0; i < 34; i++)
{
    fregion.Put("KEY--" + i, "VALUE--" + i, null);
}

object[] routingObj = new object[17];
int j = 0;
for (int i = 0; i < 34; i++)
{
    if (i % 2 == 0) continue;
    routingObj[j] = "KEY--" + i;
    j++;
}
object args0 = true;
Boolean getResult = true;
// data dependent function execution -- get function with result
Execution<object> exc =
    Generic.FunctionService.OnRegion<string, string, object>(fregion);
Generic.IResultCollector rc =
    exc.WithArgs((IGFSerializable)args0).WithFilter(
        (IGFSerializable[])routingObj).Execute(getFuncName, getResult);
object[] executeFunctionResult = rc.GetResult();

Running a Function on a Server Pool (C# .NET)

exc = Generic.FunctionService.OnServer<object>(cache);
List<object> args1 = new List<object>();
for (int i = 0; i < routingObj.Length; i++)
{
    Console.WriteLine("routingObj[{0}]={1}". i, (routingObj[i] as
        string));
    args1.Add(routingObj[i]);
}
rc = exc.WithArgs((IGFSerializable)args1).Execute(getFuncIName, getResult);
executeFunctionResult = rc.GetResult();
Console.WriteLine("on one server: result count= {0}.",
executeFunctionResult.Length);
Programming to Get Function Results

GemFire provides a default result collector. If you need special results handling, code a custom ResultsCollector implementation to replace the default provided by GemFire. Use the Execution::withCollector method to define your custom collector.

Note: This section applies only to functions that return results.

To program your client to get the results from a function, use the result collector returned from the function execution, like this:

```cpp
ResultCollectorPtr rc = FunctionService::onRegion(region)
    ->withArgs(args)
    ->withFilter(keySet)
    ->withCollector(new MyCustomResultCollector())
    .execute(Function);
CacheableVectorPtr functionResult = rc.getResult();
```

The `getResult` methods of the Pivotal GemFire default result collector block until all results are received, then return the full result set.

You can handle the results in a custom manner if you wish. To do this:

1. Write a class that extends `ResultCollector` and code the methods to handle the results as you need. The methods are of two types: one handles data and information from GemFire and populates the results set, while the other returns the compiled results to the calling application:
   a. `addResult` is called by GemFire when results arrive from the Function methods. Use `addResult` to add a single result to the ResultCollector.
   b. `endResults` is called by GemFire to signal the end of all results from the function execution.
   c. `getResult` is available to your executing application (the one that calls `Execution.execute`) to retrieve the results. This may block until all results are available.
   d. `clearResults` is called by GemFire to clear partial results from the results collector. This is used only for highly available `onRegion` functions where the calling application waits for the results. If the call fails, before GemFire retries the execution, it calls `clearResults` to ready the instance for a clean set of results.

2. Use the `Execution` object in your executing member to call `withCollector`, passing your custom collector, as shown in the example above.
Solutions and Use Cases

The function execution service provides solutions for various application use cases.

- An application that executes a server-side transaction or makes data updates using the GemFire distributed locking service.
- An application that initializes some of its components once on each server, which might be used later by executed functions.
- Initialization and startup of a third-party service, such as a messaging service.
- Any arbitrary aggregation operation that requires iteration over local data sets that can be done more efficiently through a single call to the cache server.
- Any kind of external resource provisioning that can be done by executing a function on a server.
Delta Propagation describes how deltas (updates to data) are propagated and how to implement delta propagation. It also analyzes performance limitations.

In most distributed data management systems, stored data is created once and updated frequently. Updates are sent to other members for event propagation, redundancy management, and cache consistency in general. Tracking only the changes in an updated object and sending only the updates, or deltas, mean lower network transmission costs and lower object serialization/deserialization costs. Generally, the larger your objects and the smaller the deltas, the greater the performance benefits of delta propagation. Partitioned regions generally benefit more with higher redundancy levels.

How Delta Propagation Works

Pivotal GemFire propagates object deltas using methods that you program on the client side. The methods are in the delta interface, which you implement in your cached objects’ classes.

This figure shows delta propagation for a change to an entry with key, k, and value object, v.
Figure 13: Delta Propagation

1. **get operation.** The get works as usual; the cache returns the full entry object from the local cache or, if it is unavailable there, from a server cache or from a loader.

2. **update methods.** You need to add code to the object’s update methods so that they save delta information for object updates, in addition to the work they were already doing.

3. **put operation.** The put works as usual in the local cache, using the full value, then calls `hasDelta` to check for deltas and `toDelta` to serialize the information.

4. **receipt of delta.** `fromDelta` extracts the delta information that was serialized by `toDelta` and applies it to the object in the local cache. The delta is applied directly to the existing value or to a clone, depending on how you configure it for the region.

5. **additional distributions.** As with full distributions, receiving members forward the delta according to their configurations and connections to other members. In the example, the server would forward the delta to its peers and its other clients as needed. Receiving members do not recreate the delta; `toDelta` is only called in the originating member.

## Delta Propagation API

Delta propagation uses configuration properties and a simple API to send and receive deltas.

### .NET - for C#

Your application class must implement:

- `GemStone::GemFire::Cache::IGFDelta`
Cloning

IGFDelta provides the methods, HasDelta, ToDelta, and FromDelta, which you program to report on, send, and receive deltas for your class.

Additionally, for cloning, your class must implement the standard .NET ICloneable interface and its Clone method. See Cloning.

C++

Your application must publicly derive from:

- gemfire::Delta
- gemfire::Cacheable

Delta provides the methods, hasDelta, toDelta, fromDelta, which you program to report on, send, and receive deltas for your class.

For cloning, use the clone method provided in the Delta interface. See Cloning.

Cloning

With cloning enabled, Pivotal GemFire does a deep copy of the object, using serialization. You can improve performance by implementing the appropriate clone method for your API, making a deep copy of anything to which a delta may be applied.

The goal is to significantly reduce the overhead of copying the object while still retaining the isolation needed for your deltas.

You configure delta propagation on the server side as well as client. For information on the server and delta propagation, see the Delta Propagation chapter in the GemFire User's Guide.

cloning-enabled

The cloning-enabled property is a region attributes boolean, configured in the cache.xml, that affects how fromDelta applies deltas to the local client cache. When true, the updates are applied to a clone of the value and then the clone is saved to the cache. When false, the value is modified in place in the cache. The default value is false.

Cloning can be expensive, but it ensures that the new object is fully initialized with the delta before any application code sees it.

Without cloning:

- It is possible for application code to read the entry value as it is being modified, possibly seeing the value in an intermediate, inconsistent state, with just part of the delta applied. You may choose to resolve this issue by having your application code synchronize on reads and writes.
- GemFire loses any reference to the old value because the old value is transformed in place into the new value. Because of this, your CacheListener sees the same new value returned for EntryEvent.getOldValue and EntryEvent.getNewValue.
- Exceptions thrown from fromDelta may leave your cache in an inconsistent state. Without cloning, any interruption of the delta application could leave you with some fields in your cached object changed and others unchanged. If you do not use cloning, keep this in mind when you program your error handling in your fromDelta implementation.

Enabling Cloning in cache.xml

```xml
<region name="exampleRegion">
  <region-attributes refid="CACHING_PROXY" cloning-enabled="true"
```
Implementing Delta Propagation

By default, delta propagation is enabled in your distributed system and is used for objects that implement the delta interface. You program the client-side methods to extract delta information for your entries and to apply received delta information.

For more information, see Delta Propagation API.

Prerequisites

• Study your object types and expected application behavior to determine which objects should use delta propagation. Delta propagation is not beneficial for all data and data modification scenarios. See Exceptions and Limitations.
• Decide whether to enable cloning. Cloning is disabled by default. See cloning-enabled.
  • If you enable cloning, consider providing your own implementation, to optimize performance.
  • If you do not enable cloning, be sure to synchronize your delta code.
• If you do not enable cloning, review all associated listener code for dependencies on the entry event old value. Without cloning, GemFire modifies the entry in place and so loses its reference to the old value. For delta events, the EntryEvent methods to retrieve the old and new values both return the new value.

Procedure

For every class in which you want delta propagation, implement the delta interface and update your methods to support delta propagation. Exactly how you do this depends on your application and object needs, but these steps describe the basic approach.

1. Study the object contents to decide how to handle delta changes.

   Delta propagation has the same issues of distributed concurrency control as the distribution of full objects, but on a more detailed level. Some parts of your objects may be able to change independent of one another while others may always need to change together. Send deltas large enough to keep your data logically consistent. If, for example, field A and field B depend on each other, then your delta distributions should either update both fields or neither. As with regular updates, the fewer producers you have on a data region, the lower your likelihood of concurrency issues.

2. In the application code that puts entries, put the fully populated object into the local cache. This usually means doing a get on the entry, unless you are sure it does not already exist anywhere in the distributed region.

   Even though you are planning to send only deltas, errors on the receiving end could cause GemFire to request the full object, so you must provide it to the originating put method. Do this even in empty producers, with regions configured for no local data storage.

3. Change each field’s update method to record information about the update.
The information must be sufficient for `toDelta` to encode the delta and any additional required delta information when it is invoked.

4. Write `hasDelta` to report on whether a delta is available.

5. When writing your serialization methods, `toDelta`, `fromDelta`, `toData`, `fromData`, be sure to exclude any fields used to manage delta state, which do not need to be sent.

6. Write `toDelta` to create a byte stream with the changes to the object and any other information that `fromDelta` will need to apply the changes. Before returning from `toDelta`, reset your delta state to indicate that there are no delta changes waiting to be sent.

7. Write `fromDelta` to decode the byte stream that `toDelta` creates and update the object.

8. Make sure you provide adequate synchronization to your object to maintain a consistent object state.

If you do not use cloning, you will probably need to synchronize on reads and writes to avoid reading partially written updates from the cache. This might involve `toDelta`, `fromDelta`, and other object access and update methods. Additionally, your implementation should take into account the possibility of concurrent invocations of `fromDelta` and one or more of the object's update methods.

---

**Exceptions and Limitations**

In some application scenarios, delta propagation does not show any significant performance benefits. On occasion it results in degradation. There are other limitations and exceptions related to delta propagation.

- **Errors in Delta Propagation**
- **Performance Limitations**
- **Configurations That Require Full Values**
- **General Limitations**

**Errors in Delta Propagation**

Errors in delta propagation fall into two categories based on how they are handled:

- **Problems applying the delta that can be remedied if the originating member sends the full value in place of the delta.** Your put operation does not see errors or exceptions related to the failed delta propagation. The system automatically does a full value distribution to the receiver where the problem occurs. This type of error includes:
  
  - Unavailable entry value in the receiving cache, either because the entry is missing or its value is null. In both cases, there is nothing to apply the delta to and the full value must be sent. This is most likely to occur if you destroy or invalidate your entries locally, either through application calls or through configured actions like eviction or entry expiration.
  
  - `InvalidDeltaException` thrown by `fromDelta` method, programmed by you. This exception enables you to avoid applying deltas that would violate data consistency checks or other application requirements. Throwing this exception causes GemFire to send the full value.
  
  - Any error applying the delta in a client in server-to-client propagation. The client retrieves the full value from the server.

- **Problems creating or distributing the delta that cannot be fixed by distributing the full value.** These problems are caused by errors or exceptions in `hasDelta` or `toDelta`. In these cases, your put operation fails with an exception.

**Performance Limitations**

Consider the following situations in which delta propagation adversely affects performance:

- **Added costs of deserializing your objects to apply deltas.** Applying a delta requires the entry value to be deserialized. Once this is done, the object is stored back in the cache in deserialized form. This
aspect of delta propagation only negatively impacts your system if your objects are not already being deserialized for other reasons, such as for indexing and querying or for listener operations. Once stored in deserialized form, there are reserialization costs for operations that send the object outside of the member, like values sent in response to netSearch or client requests, and storage to disk. The more operations that require reserialization, the higher the overhead of deserializing the object.

- **Cloning.** Cloning can affect performance. Not using cloning is risky, however, as you are modifying cached values in place. Make sure you synchronize your entry access to keep your cache from becoming inconsistent.

- **Problems applying the delta that cause the system to go back to the originator for the full entry value.** In this scenario, the overall operation costs more than sending the full entry value in the first place. The problem can be exacerbated if your delta is sent to a number of recipients, all or most of them request a full value, and the full value send requires the object to be serialized.

- **Disk I/O costs associated with overflow regions.** If you use eviction with overflow to disk, on-disk values must be brought into memory in order to apply the delta. This is much more costly than removing the reference to the disk copy, as you would do with a full value distribution into the cache.

### Configurations That Require Full Values

GemFire clients can always send deltas to the servers, and servers can usually sent deltas to clients. The following configurations require the servers to send full values to the clients, instead of deltas:

- When the client’s `gemfire.properties` setting `conflate-events` is set to `true`, the servers send full values for all regions.

- When the server region attribute `enable-subscription-conflation` is set to `true` and the client `gemfire.properties` setting `conflate-events` is set to `server`, the servers send full values for the region.

- Servers send full values to client regions that are configured to not cache data—with the `PROXY RegionShortcut` in their region attributes `refid`.

To use the delta propagation feature, all updates on a key in a region must have value types that implement the `Delta` interface. You cannot mix object types for an entry key where some of the types implement delta and some do not. This is because, when a type implementing the delta interface is received for an update, the existing value for the key is cast to a `Delta` type to apply the received delta.

### General Limitations

Sometimes `fromDelta` cannot be invoked because there is no object to apply the delta to in the receiving cache. When this happens, the system sends the full value. There are two possible scenarios:

- If the system can determine beforehand that the receiver does not have a local copy, it sends the initial message with the full value. This is possible when regions are configured with no local data storage, as when you are using them to provide data update information to listeners.

- In less obvious cases, as when an entry has been locally deleted, first the delta is sent, then the receiver requests a full value and that is sent. Whenever the full value is received, any further distributions to the receiver’s peers or clients uses the full value.

GemFire does not propagate deltas for:

- Transactional commit on the server
- The `putAll` operation
- Server VMs running GemFire versions 6.0 and earlier
Examples of Delta Propagation

Examples describe delta propagation operations and provide implementation code for C# .NET and C++.

Delta Propagation in the Client/Server

In this example, the feeder client is connected to the first server, and the receiver client is connected to the second. The servers are peers to each other.

![Diagram of Delta Propagation in the Client/Server](image)

These are the main operations shown in the example:

1. In the Feeder client, the application updates the entry object and puts the entry. In response to the `put`, Pivotal GemFire calls `hasDelta`, which returns true.
2. GemFire calls `toDelta` and forwards the extracted delta to the server. If `hasDelta` returned false, GemFire would distribute the full entry value.
3. In Server1, GemFire applies the delta to the cache, distributes the received delta to the server’s peers, and forwards it to any other clients with interest in the entry (there are no other clients to Server1 in this example).
4. In Server2, GemFire applies the delta to the cache and forwards it to its interested clients, which in this case is just Receiver client.

Client Example Files

These example files, from the product quickstart examples, show the basic approach to programming a delta propagation implementation for an entry value object, named `DeltaExample` in this example.

XML file used for the examples

```xml
<cache>
  <region name="root" refid="CACHING_PROXY">
    <region-attributes cloning-enabled="true" pool-name="examplePool"/>
  </region>
</cache>

<pool name="examplePool" subscription-enabled="true" server-group="ServerGroup1">
  <locator host="localhost" port="34756"/>
```
using System;

{
    public class DeltaExample : IGFDelta, IGFSerializable, ICloneable
    {
        // data members
        private Int32 m_field1;
        private Int32 m_field2;
        private Int32 m_field3;

        // delta indicators
        private bool m_f1set;
        private bool m_f2set;
        private bool m_f3set;

        public DeltaExample(Int32 field1, Int32 field2, Int32 field3)
        {
            m_field1 = field1;
            m_field2 = field2;
            m_field3 = field3;
            reset();
        }

        public DeltaExample()
        {
            reset();
        }

        public DeltaExample(DeltaExample copy)
        {
            m_field1 = copy.m_field1;
            m_field2 = copy.m_field2;
            m_field3 = copy.m_field3;
            reset();
        }

        private void reset()
        {
            m_f1set = false;
            m_f2set = false;
            m_f3set = false;
        }

        public Int32 getField1()
        {
            return m_field1;
        }

        public void setField1(Int32 val)
        {
            lock(this)
            {
                m_field1 = val;
                m_f1set = true;
            }
        }

        public bool HasDelta()
        {
            return m_f1set || m_f2set || m_f3set;
        }
    }
}
public void ToDelta(DataOutput DataOut) {
    lock(this) {
        DataOut.WriteBoolean(m_f1set);
        if (m_f1set) {
            DataOut.WriteInt32(m_field1);
            // REPEAT FOR OTHER FIELDS
            reset();
        }
    }
}

public void FromDelta(DataInput DataIn) {
    lock(this) {
        m_f1set = DataIn.ReadBoolean();
        if (m_f1set) {
            m_field1 = DataIn.ReadInt32();
            // REPEAT FOR OTHER FIELDS
        }
    }
}

public void ToData(DataOutput DataOut) {
    DataOut.WriteInt32(m_field1);
    DataOut.WriteInt32(m_field2);
    DataOut.WriteInt32(m_field3);
}

public IGFSerializable FromData(DataInput DataIn) {
    m_field1 = DataIn.ReadInt32();
    m_field2 = DataIn.ReadInt32();
    m_field3 = DataIn.ReadInt32();
    return this;
}

public UInt32 ClassId {
    get {
        return 0x02;
    }
}

public UInt32 ObjectSize {
    get {
        UInt32 objectSize = 0;
        return objectSize;
    }
}

public static IGFSerializable create() {
    return new DeltaExample();
}

public Object Clone() {
    return new DeltaExample(this);
}
Delta Example Implementation (C++)

```cpp
#ifndef __Delta_Example__
#define __Delta_Example__

#include <gfcpp/GemfireCppCache.hpp>

using namespace gemfire;

class DeltaExample: public Cacheable, public Delta {

private:
    // data members
    int32_t m_field1;
    int32_t m_field2;
    int32_t m_field3;

    // delta indicators
    mutable bool m_f1set;
    mutable bool m_f2set;
    mutable bool m_f3set;

public:
    DeltaExample(int32_t field1, int32_t field2, int32_t field3) :
        m_field1(field1), m_field2(field2), m_field3(field3)
    {
        reset();
    }

    DeltaExample()
    {
        reset();
    }

    DeltaExample(DeltaExample * copy)
    {
        m_field1 = copy->m_field1;
        m_field2 = copy->m_field2;
        m_field3 = copy->m_field3;
        reset();
    }

    void reset() const
    {
        m_f1set = false;
        m_f2set = false;
        m_f3set = false;
    }

    int getField1()
    {
        return m_field1;
    }

    // REPEAT FOR OTHER FIELDS

    void setField1(int val)
    {
        lock();
        m_field1 = val;
        m_f1set = true;
        unlock();
    }

    // REPEAT FOR OTHER FIELDS

    virtual bool hasDelta()
```
{  
    return m_f1set || m_f2set || m_f3set;
}

virtual void toDelta(DataOutput& out) const
{
    lock();
    out.writeBoolean(m_f1set);
    if (m_f1set)
    {
        out.writeInt(m_field1);
    }
    // REPEAT FOR OTHER FIELDS
    reset();
    unlock();
}

virtual void fromDelta(DataInput& in)
{
    lock();
    in.readBoolean(&m_f1set);
    if (m_f1set)
    {
        in.readInt(&m_field1);
    }
    // REPEAT FOR OTHER FIELDS
    reset();
    unlock();
}

virtual void toData(DataOutput& output) const
{
    lock();
    output.writeInt(m_field1);
    output.writeInt(m_field2);
    output.writeInt(m_field3);
    unlock();
}

virtual Serializable* fromData(DataInput& input)
{
    lock();
    input.readInt(&m_field1);
    input.readInt(&m_field2);
    input.readInt(&m_field3);
    unlock();
    return this;
}

virtual int32_t classId() const
{
    return 2;
}

virtual uint32_t objectSize() const
{
    return 0;
}

DeltaPtr clone()
{
    return DeltaPtr(new DeltaExample(this));
}

virtual ~DeltaExample()
{ }

static Serializable* create()
{
    return new DeltaExample();
}

void lock() const { /* add your platform dependent synchronization code here */ }

void unlock() const { /* add your platform dependent synchronization code here */ }
#endif
This chapter provides a set of programming examples to help you understand the Pivotal GemFire native client API.

### Declaring a Native Client Region

The following example shows how to declare a native client region in a cache.xml file.

```xml
<cache>
  <region name = "root1" >
    <region-attributes refid="CACHING_PROXY" pool-name="poolName1"/>
  </region>
  <region name = "root2" >
    <region-attributes refid="PROXY" pool-name="poolName2"/>
  </region>
  <pool name="poolName1" subscription-enabled="true">
    <server host="localhost" port="40404" />
  </pool>
  <pool name="poolName2" subscription-enabled="true">
    <server host="localhost" port="40404" />
  </pool>
</cache>
```

**Figure 15: Declaring a Native Client Region Using cache.xml**

- The pool defines a list of cache servers that the native client region can communicate with.
- The **CACHING_PROXY** setting causes the client region to cache data and to communicate with the servers. The **PROXY** setting causes the client region to communicate with the servers, but cache no data.
- The region **subscription-enabled** property, if true, indicates that the client should receive data updates when server data changes.
- Native clients do not specify cache loaders or writers, which are provided by the server.

### API Programming Example – C#

This C# programming code in the next example demonstrates how to use two or more clients sharing a distributed region in a GemFire cache.

```csharp
using System;

// Use the GemFire namespace

```


```csharp
// The BasicOperations QuickStart example.

class BasicOperations
{
    static void Main(string[] args)
    {
        try
        {
            // Create a GemFire Cache.
            CacheFactory cacheFactory = CacheFactory.CreateCacheFactory(null);
            Cache cache = cacheFactory.SetSubscriptionEnabled(true).Create();
            Console.WriteLine("Created the GemFire Cache");
            RegionFactory regionFactory = cache.CreateRegionFactory(RegionShortcut.CACHING_PROXY);
            Region region = regionFactory.Create("exampleRegion");
            Console.WriteLine("Created the Region Programmatically.");
            // Put an Entry (Key and Value pair) into the Region using the
            // direct/shortcut method.
            region.Put("Key1", "Value1");
            Console.WriteLine("Put the first Entry into the Region");
            region.Put(new CacheableInt32(123), new CacheableString("123"));
            region.Put(new CacheableInt32(123), new CacheableString("123"));
            Console.WriteLine("Put the second Entry into the Region");
            if (IntPtr.Size == 8) // Are we a 64 bit process?
            {
                Char ch = 'A';
                string text = new string(ch, 1024 * 1024);
                for (int item = 0; item < (5 * 1024 /* 5 GB */); item++)
                {
                    region.LocalPut(item, text);
                }
                Console.WriteLine("Put over 4 GB data locally");
            }
            // Get Entries back out of the Region.
            IGFSerializable result1 = region.Get("Key1");
            Console.WriteLine("Obtained the first Entry from the Region");
            IGFSerializable result2 = region.Get(key);
            Console.WriteLine("Obtained the second Entry from the Region");
            // Invalidate an Entry in the Region.
            region.Invalidate("Key1");
            Console.WriteLine("Invalidated the first Entry in the Region");
            // Destroy an Entry in the Region.
            region.Destroy(key);
            Console.WriteLine("Destroyed the second Entry in the Region");
            // Close the GemFire Cache.
            cache.Close();
            Console.WriteLine("Closed the GemFire Cache");
        }
        // An exception should not occur
    }
}
```
The next example uses the C++ API to implement a cache loader, which is generally used to retrieve data from an outside source.

```cpp
CacheablePtr TestCacheLoader::load(
    const RegionPtr& region,
    const CacheableKeyPtr& key,
    const UserDataPtr& aCallbackArgument)
{
    m_bInvoked = true;
    printf( "CacheLoader.load : %s\n", printEvent(region, key,
        aCallbackArgument).c_str());
    CacheablePtr value = NULLPTR;
    try {
        value = region->get(key, aCallbackArgument);
    } catch(Exception& ex) {
        fprintf(stderr, "Exception in TestCacheCallback::printEvent [%s]\n",
            ex.getMessage());
    }
    if (value != NULLPTR) {
        printf( "Loader found value: ");
        std::string formatValue = printEntryValue(value);
        printf( "%s\n", formatValue.c_str());
    } else {
        printf( " Loader did not find a value");
    }
    return value;
}
```

**Figure 17: Implementing a Cache Loader (C++)**

**Data Serialization Examples**

This section contains data serialization examples for C++, C#, and Java.
C++ Serialization Example

class User
: public Serializable
{
  private:
    std::string name;
    int32_t userId;
    ExampleObject *eo;
  public:
  User( std::string name, int32_t userId )
    : name( name ), userId( userId )
  {
    eo = new ExampleObject(this->userId);
  }
  ~User() {
    if (eo != NULL) delete eo;
    eo = NULL;
  }
  User() {
    name = "";
    userId = 0;
    eo = new ExampleObject(userId);
  }
  User( const char *strfmt, char delimiter )
  {
    std::string userId_str;
    std::string sValue(strfmt);
    std::string::size_type pos1 = sValue.find_first_of(delimiter);
    std::string::size_type pos2;
    if (pos1 == std::string::npos) {
      userId_str = sValue;
      name = sValue;
    } else {
      userId_str = sValue.substr(0, pos1);
      pos2 = sValue.find(delimiter, pos1+1);
      int len;
      if (pos2==std::string::npos) {
        len = sValue.length()-pos1;
      } else {
        len = pos2-pos1;
      }
      name = sValue.substr(pos1+1, len);
    }
    userId = (int32_t)atoi(userId_str.c_str);
    eo = new ExampleObject(userId_str);
  }
  CacheableStringPtr toString() const
  {
    CacheableStringPtr eo_str = eo->toString();
    char user_id_str[128];
    sprintf(user_id_str,"User: %d", userId);
    std::string sValue = std::string(user_id_str) + "," + name + "\n"
     + std::string(eo_str->asChar());
    return CacheableString::create( sValue.c_str() );
  }
  int32_t getUserId( )
  {
  
  }
return userId;
}

std::string getName( )
{
    return name;
}

ExampleObject *getEO()
{
    return eo;
}

void setEO(ExampleObject *eObject)
{
    eo = eObject;
}

// Add the following for the Serializable interface
// Our TypeFactoryMethod

static Serializable* createInstance( )
{
    return new User(std::string("gester"), 123);
}

int32_t classId( ) const
{
    return 0x2d; // 45
}

void toData( DataOutput& output ) const
{
    output.writeASCII( name.c_str(), name.size() );
    output.writeInt( userId );
    eo->toData(output);
}

uint32_t objectSize( ) const
{
    return ( sizeof(char) * ( name.size() + 1 ) ) +
            sizeof(User) + eo->objectSize();
}

Serializable* fromData( DataInput& input )
{
    char *readbuf;
    input.readASCII( &readbuf );
    name = std::string(readbuf);
    input.freeUTFMemory( readbuf );
    input.readInt( &userId );
    eo->fromData(input);
    return this;
};

Figure 18: Implementing an Embedded Object (C++)

class ExampleObject
    : public Serializable
{
private:
    double double_field;
    float float_field;
    long long_field;
    int int_field;
    short short_field;
    std::string string_field;
    std::vector<std::string> string_vector;
public:
```cpp
ExampleObject() {
    double_field = 0.0;
    float_field = 0.0;
    long_field = 0;
    int_field = 0;
    short_field = 0;
    string_field = "";
    string_vector.clear();
}
~ExampleObject() {
}
ExampleObject(int id) {
    char buf[64];
    sprintf(buf, "%d", id);
    std::string sValue(buf);
    int_field = id;
    long_field = int_field;
    short_field = int_field;
    double_field = (double)int_field;
    float_field = (float)int_field;
    string_field = sValue;
    string_vector.clear();
    for (int i=0; i<3; i++) {
        string_vector.push_back(sValue);
    }
}
ExampleObject(std::string sValue) {
    int_field = atoi(sValue.c_str());
    long_field = int_field;
    short_field = int_field;
    double_field = (double)int_field;
    float_field = (float)int_field;
    string_field = sValue;
    string_vector.clear();
    for (int i=0; i<3; i++) {
        string_vector.push_back(sValue);
    }
}
CacheableStringPtr toString() const {
    char buf[1024];
    std::string sValue = "ExampleObject: ";
    sprintf(buf,"%f(double),%f(double),%ld(long),%d(int),%d(short),",
    double_field,float_field,long_field,int_field,short_field);
    sValue += std::string(buf) + string_field + "(string),";
    if (string_vector.size() >0) {
        sValue += "[";
        for (unsigned int i=0; i<string_vector.size(); i++) {
            sValue += string_vector[i];
            if (i != string_vector.size()-1) {
                sValue += ",";
            }
        }
        sValue += "]";string_vector});
    }
    return CacheableString::create( sValue.c_str() );
}
double getDouble_field() {
    return double_field;
}
float getFloat_field() {
    return float_field;
}
long getLong_field() {
    return long_field;
}
int getInt_field() {
    return int_field;
}
short getShort_field() {
    return short_field;
}
```
std::string & getString_field() {  
    return string_field;
}
std::vector<std::string> & getString_vector() {  
    return string_vector;
}

// Add the following for the Serializable interface
// Our TypeFactoryMethod
static Serializable* createInstance() {  
    return new ExampleObject();
}

int32_t classId() const  
{  
    return 0x2e; // 46
}

bool operator == ( const Serializable& other ) const {  
    const ExampleObject& otherEO = static_cast<const ExampleObject&>( other );  
    return ( 0 == strcmp( otherEO.toString()->asChar(), toString()->asChar() ) );
}

uint32_t hashcode() const {  
    return int_field;
}

uint32_t objectSize() const {  
    uint32_t objectSize = sizeof(ExampleObject);  
    objectSize += sizeof(char) * ( string_field.size() + 1 );  
    size_t itemCount = string_vector.size();  
    for( size_t idx = 0; idx < itemCount; idx++ ) {  
        // copy each string to the serialization buffer, including the null  
        // terminating character at the end of the string.
        objectSize += sizeof(char) * ( string_vector[idx].size() + 1 );
    }
    return objectSize;
}

void toData( DataOutput& output ) const {  
    output.writeDouble( double_field );  
    output.writeFloat( float_field );  
    output.writeInt( (int64_t)long_field );  
    output.writeInt( (int32_t)int_field );  
    output.writeInt( (int16_t)short_field );  
    output.writeASCII( string_field.c_str(), string_field.size() );  
    size_t itemCount = string_vector.size();  
    for( size_t idx = 0; idx < itemCount; idx++ ) {  
        // copy each string to the serialization buffer, including the null  
        // terminating character at the end of the string.
        output.writeASCII( string_vector[idx].c_str(), string_vector[idx].size() );
    }
}

Serializable* fromData( DataInput& input ) {  
    char *readbuf;  
    input.readDouble( &double_field );  
    input.readFloat( &float_field );  
    input.readInt( (int64_t *)&long_field );  
    input.readInt( (int32_t *)&int_field );  
    input.readInt( (int16_t *)&short_field );  
    input.readUTFMemory( readbuf );  
    string_field = std::string( readbuf );  
    input.freeUTFMemory( readbuf );
    string_vector.clear();  
    input.readInt( (int32_t *)&itemCount );  
    for( int32_t idx = 0; idx < itemCount; idx++ ) {  
        //
// read from serialization buffer into a character array
input.readASCII( &readbuf );
// and store in the history list of strings.
string_vector.push_back( readbuf );
input.freeUTFMemory( readbuf );
}
return this;
}
typedef SharedPtr<ExampleObject> ExampleObjectPtr;

Figure 19: Implementing Complex Data Types (C++)
C# Serialization Example
This example shows how to implement a user-defined Serializable object.

class User : IGFSerializable
{
    private string m_name;
    private int m_userId;
    ExampleObject m_eo;

    public User(string name, int userId)
    {
        m_name = name;
        m_userId = userId;
        m_eo = new ExampleObject();
    }

    public User()
    {
        m_name = string.Empty;
        m_userId = 0;
        m_eo = new ExampleObject();
    }

    public int UserId
    {
        get
        {
            return m_userId;
        }
    }

    public string Name
    {
        get
        {
            return m_name;
        }
    }

    public ExampleObject EO
    {
        get
        {
            return m_eo;
        }
        set
        {
            m_eo = value;
        }
    }

    public override string ToString()
    {
        return string.Format("User: {0}, {1}\n{2}", m_userId, m_name, m_eo.ToString());
    }

    // Our TypeFactoryMethod
    public static IGFSerializable CreateInstance()
    {
        return new User();
    }

    #region IGFSerializable Members

}
public UInt32 ClassId()
{
    get
    {
        return 45; // must match Java
    }
}

public IGFSerializable FromData(DataInput input)
{
    m_name = input.ReadUTF();
    m_userId = input.ReadInt32();
    m_eo.FromData(input);
    return this;
}

public void ToData(DataOutput output)
{
    output.writeUTF(m_name);
    output.writeInt32(m_userId);
    eo.ToData(output);
}

#endregion

Figure 20: Implementing a User-Defined Serializable Object (C# .NET)
Java Serialization Example

Implementing an Embedded Object (Java)

```java
public class User implements DataSerializable {
    private String name;
    private int userId;
    private ExampleObject eo;
    static {
        Instantiator.register(new Instantiator(User.class, (byte)45) {
            public DataSerializable newInstance() {
                return new User();
            }
        });
    }

    /**
     * Creates an "empty" User whose contents are filled in by
     * invoking its toData() method
     */
    private User() {
        this.name = "";
        this.userId = 0;
        this.eo = new ExampleObject(0);
    }

    public User(String name, int userId) {
        this.name = name;
        this.userId = userId;
        this.eo = new ExampleObject(userId);
    }

    public void setEO(ExampleObject eo) {
        this.eo = eo;
    }

    public ExampleObject getEO() {
        return eo;
    }

    public void toData(DataOutput out) throws IOException {
        out.writeUTF(this.name);
        out.writeInt(this.userId);
        eo.toData(out);
    }

    public void fromData(DataInput in) throws IOException, ClassNotFoundException {
        this.name = in.readUTF();
        this.userId = in.readInt();
        this.eo.fromData(in);
    }

    public int getUserId() {
        return userId;
    }

    public String getName() {
        return name;
    }
}
```
public boolean equals(User o) {
    if (!this.name.equals(o.name)) return false;
    if (this.userId != o.userId) return false;
    if (!this.eo.equals(o.eo)) return false;
    return true;
}

Implementing Complex Data Types (Java)

public class ExampleObject implements DataSerializable {
    private double double_field;
    private long long_field;
    private float float_field;
    private int int_field;
    private short short_field;
    private java.lang.String string_field;
    private Vector string_vector;
    static {
        Instantiator.register(new Instantiator(ExampleObject.class, (byte) 46) {
            public DataSerializable newInstance() {
                return new ExampleObject();
            }
        });
    }
    public ExampleObject() {
        this.double_field = 0.0D;
        this.long_field = 0L;
        this.float_field = 0.0F;
        this.int_field = 0;
        this.short_field = 0;
        this.string_field = null;
        this.string_vector = null;
    }
    public ExampleObject(int id) {
        this.int_field = id;
        this.string_field = String.valueOf(id);
        this.short_field = Short.parseShort(string_field);
        this.double_field = Double.parseDouble(string_field);
        this.float_field = Float.parseFloat(string_field);
        this.long_field = Long.parseLong(string_field);
        this.string_vector = new Vector();
        for (int i=0; i<3; i++) {
            this.string_vector.addElement(string_field);
        }
    }
    public ExampleObject(String id_str) {
        this.int_field = Integer.parseInt(id_str);
        this.string_field = id_str;
        this.short_field = Short.parseShort(string_field);
        this.double_field = Double.parseDouble(string_field);
        this.float_field = Float.parseFloat(string_field);
        this.long_field = Long.parseLong(string_field);
        this.string_vector = new Vector();
        for (int i=0; i<3; i++) {
            this.string_vector.addElement(string_field);
        }
    }
    public double getDouble_field() {
        return this.double_field;
    }
    public void setDouble_field( double double_field ) {
        this.double_field = double_field;
    }
    public long getLong_field() {

return this.long_field;
}
public void setLong_field( long long_field ) {
    this.long_field = long_field;
}
public float getFloat_field( ) {
    return this.float_field;
}
public void setFloat_field( float float_field ) {
    this.float_field = float_field;
}
public int getInt_field( ) {
    return this.int_field;
}
public void setInt_field( int int_field ) {
    this.int_field = int_field;
}
public short getShort_field( ) {
    return this.short_field;
}
public void setShort_field( short short_field ) {
    this.short_field = short_field;
}
public java.lang.String getString_field( ) {
    return this.string_field;
}
public void setString_field( java.lang.String string_field ) {
    this.string_field = string_field;
}
public Vector getString_vector( ) {
    return this.string_vector;
}
public void setString_vector( Vector string_vector ) {
    this.string_vector = string_vector;
}
public void toData(DataOutput out) throws IOException {
    out.writeDouble(double_field);
    out.writeFloat(float_field);
    out.writeLong(long_field);
    out.writeInt(int_field);
    out.writeShort(short_field);
    out.writeUTF(string_field);
    out.writeInt(string_vector.size());
    for (int i = 0; i < string_vector.size(); i++)
    {
        out.writeUTF((String)string_vector.elementAt(i));
    }
}
public void fromData(DataInput in) throws IOException, ClassNotFoundException {
    this.double_field = in.readDouble();
    this.float_field = in.readFloat();
    this.long_field = in.readLong();
    this.int_field = in.readInt();
    this.short_field = in.readShort();
    this.string_field = in.readUTF();
    this.string_vector = new Vector();
    int size = in.readInt();
    for (int i = 0; i < size; i++)
    {
        String s = in.readUTF();
        string_vector.add(i, s);
    }
}
public boolean equals(ExampleObject o) {
    if (this.double_field != o.double_field) return false;
    if (this.float_field != o.float_field) return false;
    if (this.long_field != o.long_field) return false;
if (this.int_field != o.int_field) return false;
if (this.short_field != o.short_field) return false;
if (!this.string_field.equals(o.string_field)) return false;
if (!this.string_vector.equals(o.string_vector)) return false;
return true;
Interoperability of Language Classes and Types

Interoperability of Language Classes and Types provides a table that maps C++ class methods to corresponding .NET class methods and a table that maps Java types to .NET types.

C++ Class to .NET Class Mappings

Wherever the native C++ class methods use pass-by-reference semantics to return data, the corresponding .NET methods shown in the following table return the object instead of using pass-by-reference semantics.

Table 24: C++ Class to .NET Class Mappings

<table>
<thead>
<tr>
<th>C++ Class</th>
<th>.NET Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gemfire::AttributesFactory</code></td>
<td><code>Sealed class AttributesFactory</code></td>
</tr>
<tr>
<td><code>gemfire::AttributesMutator</code></td>
<td><code>Sealed class AttributesMutator</code></td>
</tr>
<tr>
<td><code>gemfire::Cache</code></td>
<td><code>Sealed class Cache</code></td>
</tr>
<tr>
<td><code>gemfire::Cacheable</code></td>
<td><code>Interface IPdxSerializable or interface IGFSerializable</code></td>
</tr>
<tr>
<td><code>gemfire::CacheableBytes</code></td>
<td><code>Byte[] or ArrayList&lt;Byte&gt;</code></td>
</tr>
<tr>
<td><code>gemfire::CacheableInt32</code></td>
<td><code>Int32</code></td>
</tr>
<tr>
<td><code>gemfire::CacheableString</code></td>
<td><code>String</code></td>
</tr>
<tr>
<td><code>gemfire::CacheableKey</code></td>
<td><code>You can use any type that implements hashcode and equals. The generic .NET built-in types are all suitable.</code></td>
</tr>
<tr>
<td><code>gemfire::CacheListener</code></td>
<td><code>Interface ICacheListener</code></td>
</tr>
<tr>
<td><code>gemfire::CacheLoader</code></td>
<td><code>Interface ICacheLoader plus static class CacheLoader</code></td>
</tr>
<tr>
<td><code>gemfire::CacheWriter</code></td>
<td><code>Interface class ICacheWriter</code></td>
</tr>
<tr>
<td><code>gemfire::CacheFactory</code></td>
<td><code>Sealed class CacheFactory</code></td>
</tr>
<tr>
<td>C++ Class</td>
<td>.NET Class</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>class gemfire::DataInput</td>
<td>With IPdxSerializable, IPdxReader.</td>
</tr>
<tr>
<td></td>
<td>With IGFSerializable, sealed class DataInput.</td>
</tr>
<tr>
<td>class gemfire:: DataOutput</td>
<td>With IPdxSerializable, IPdxWriter.</td>
</tr>
<tr>
<td></td>
<td>With IGFSerializable, sealed class DataOutput.</td>
</tr>
<tr>
<td>class gemfire:: DiskPolicyType</td>
<td>enum DiskPolicyType plus static class DiskPolicy containing convenience methods for DiskPolicyType enumeration</td>
</tr>
<tr>
<td>class gemfire:: DistributedSystem</td>
<td>Sealed class DistributedSystem</td>
</tr>
<tr>
<td>class gemfire:: EntryEvent</td>
<td>Sealed class EntryEvent</td>
</tr>
<tr>
<td>class gemfire:: Exception</td>
<td>Class GemfireException</td>
</tr>
<tr>
<td>all other exceptions deriving from gemfire:: Exception</td>
<td>Corresponding exceptions deriving from GemfireException</td>
</tr>
<tr>
<td>class gemfire:: ExpirationAction</td>
<td>enum ExpirationAction plus static class Expiration containing convenience methods for ExpirationAction enumeration</td>
</tr>
<tr>
<td>class gemfire:: Log</td>
<td>Static class Log. The native Log::log method is mapped to Log.Write to avoid the conflict with the class name which is reserved for the constructors of Log class. The various loglevel Throw or Catch methods are not implemented, since they are redundant to Log::Log, Log::LogThrow, and Log::LogCatch methods that take LogLevel as a parameter.</td>
</tr>
<tr>
<td>enum gemfire:: MemberType</td>
<td>enum MemberType</td>
</tr>
<tr>
<td>abstract class gemfire:: PersistanceManager</td>
<td>Not provided. You can register a C++ implementation using AttributesFactory. SetPersistanceManager but you cannot implement a new one in .NET</td>
</tr>
<tr>
<td>class gemfire:: Properties</td>
<td>Sealed class Properties</td>
</tr>
<tr>
<td>class gemfire:: Properties::Visitor</td>
<td>Delegate PropertiesVisitor</td>
</tr>
<tr>
<td>abstract class gemfire:: Region</td>
<td>Class IRegion</td>
</tr>
<tr>
<td>class gemfire:: RegionAttributes</td>
<td>Sealed class RegionAttributes</td>
</tr>
<tr>
<td>class gemfire:: ScopeType</td>
<td>enum ScopeType plus static class Scope containing convenience methods for ScopeType enumeration</td>
</tr>
</tbody>
</table>
### Interoperability of Language Classes and Types

<table>
<thead>
<tr>
<th>C++ Class</th>
<th>.NET Class</th>
</tr>
</thead>
</table>
| **abstract class gemfire:: Serializable** | Two options:  
  - Interface IPdxSerializable  
  - Interface IGFSerializable plus wrapper  
  Serializable class for native Serializable and UserData objects. The native toString method is not provided, since the ToString method of the base object class provides the same functionality. |
| **class gemfire:: SystemProperties** | Sealed class SystemProperties |
| **class gemfire:: UserData** | Two options:  
  - Interface IPdxSerializable  
  - Interface IGFSerializable |
| **class gemfire:: VectorT<T>** | Array of the given type, such as T[] |
### Interoperability of C++ Types When Using PDX Serialization

This topic table lists the mapping between C++ types and other language types when using PDX serialization.

In addition, the table lists which PdxReader and PdxWriter C++ APIs to use when serializing and deserializing the types.

<table>
<thead>
<tr>
<th>C++ Type</th>
<th>.NET Type</th>
<th>Java Type</th>
<th>PdxReader/PdxWriter API</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheableHashTable</td>
<td>System::Collections::Hashtable</td>
<td>java.util.Hashtable</td>
<td>readObject/writeObject</td>
</tr>
<tr>
<td>CacheableHashMap</td>
<td>System::Collections::Generic::IDictionary&lt;Object&gt;</td>
<td>java.util.HashMap</td>
<td>readObject/writeObject</td>
</tr>
<tr>
<td>CacheableVector</td>
<td>System::Collections::ArrayList</td>
<td>java.util.Vector</td>
<td>readObject/writeObject</td>
</tr>
<tr>
<td>CacheableArrayList</td>
<td>System::Collections::Generic::IList&lt;Object&gt;</td>
<td>java.util.ArrayList</td>
<td>readObject/writeObject</td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
<td>boolean</td>
<td>readBoolean/writeBoolean</td>
</tr>
<tr>
<td>int8_t</td>
<td>sbyte</td>
<td>Byte</td>
<td>readByte/writeByte</td>
</tr>
<tr>
<td>wchar_t/char¹</td>
<td>Char</td>
<td>Char</td>
<td>readChar/writeChar</td>
</tr>
<tr>
<td>wchar_t*/char*¹</td>
<td>string</td>
<td>string</td>
<td>readString/writeString</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
<td>double</td>
<td>readDouble/writeDouble</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>float</td>
<td>readFloat/writeFloat</td>
</tr>
<tr>
<td>int16_t</td>
<td>short</td>
<td>short</td>
<td>readShort/writeShort</td>
</tr>
<tr>
<td>int32_t</td>
<td>Int32/int</td>
<td>int</td>
<td>readInt/writeInt</td>
</tr>
<tr>
<td>int64_t</td>
<td>Int64/long</td>
<td>long</td>
<td>readLong/writeLong</td>
</tr>
<tr>
<td>int8_t*</td>
<td>System.Byte<a href="System.SByte%5B%5D??"></a></td>
<td>Byte[]²</td>
<td>readByteArray/writeByteArray</td>
</tr>
<tr>
<td>double*</td>
<td>System.Double[]</td>
<td>Double[]</td>
<td>readDoubleArray/writeDoubleArray</td>
</tr>
<tr>
<td>float*</td>
<td>System.float[]</td>
<td>Float[]</td>
<td>readFloatArray/writeFloatArray</td>
</tr>
<tr>
<td>CacheableHashSet</td>
<td>CacheableHashSet</td>
<td>java.util.HashSet</td>
<td>readObject/writeObject</td>
</tr>
<tr>
<td>CacheableLinkedHashSet</td>
<td>CacheableLinkedHashSet</td>
<td>java.util.LinkedHashSet</td>
<td>readObject/writeObject</td>
</tr>
<tr>
<td>int16_t*</td>
<td>System.Int16[]</td>
<td>Short[]</td>
<td>readShortArray/writeShortArray</td>
</tr>
<tr>
<td>int32_t*</td>
<td>System.Int32[]</td>
<td>Int[]</td>
<td>readIntArray/writeIntArray</td>
</tr>
<tr>
<td>int64_t*</td>
<td>System.Int64[]</td>
<td>Long[]</td>
<td>readLongArray/writeLongArray</td>
</tr>
<tr>
<td>C++ Type</td>
<td>.NET Type</td>
<td>Java Type</td>
<td>PdxReader/PdxWriter API</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>-----------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>bool*</td>
<td>System.Boolean[]</td>
<td>Boolean[]</td>
<td>readBooleanArray/writeBooleanArray</td>
</tr>
<tr>
<td>wchar_t*/char*¹</td>
<td>System.Char[]</td>
<td>char[]</td>
<td>readCharArray/writeCharArray</td>
</tr>
<tr>
<td>enum³</td>
<td>enum</td>
<td>Enum</td>
<td>readObject/writeObject</td>
</tr>
<tr>
<td>int8_t**</td>
<td>byte[]/Sbyte[]</td>
<td>Byte[]</td>
<td>readArrayOfByteArrays/writeArrayOfByteArrays</td>
</tr>
<tr>
<td>wchar_t*/char*¹</td>
<td>System.String[]</td>
<td>String[]</td>
<td>readStringArray/writeStringArray</td>
</tr>
<tr>
<td>CacheableDate</td>
<td>System.DateTime⁴</td>
<td>java.util.date</td>
<td>readDate/writeDate</td>
</tr>
<tr>
<td>CacheableObjectArray</td>
<td>object[]/System.Object[]</td>
<td>Object[]</td>
<td>readObjectArray/writeObjectArray</td>
</tr>
<tr>
<td>Cacheable/Serializable</td>
<td>object/System.Object</td>
<td>Object</td>
<td>readObject/writeObject</td>
</tr>
</tbody>
</table>

¹C++ allows unicode and non-unicode characters, so C++ PDX will support both wchar_t/*char*.
²Using Pdx we use sByte only as java byte is signed. But for DataSerializable we have kept Byte[] array only as this is use as data container.
³C++ allows explicit setting of ordinal number, but it is up to the user to map java enumName with that of C++ enumName. See Using C++ Enum Type with PDX Serialization.
⁴Now we put UTC time from .NET.

Java to .NET Type Mapping Table

The following table provides a mapping between Java and .NET types.

Table 25: Java types and .NET types

<table>
<thead>
<tr>
<th>Java Type</th>
<th>.NET Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>instances of PdxSerializable</td>
<td>.NET class of same name</td>
</tr>
<tr>
<td>instances of PdxInstance</td>
<td>.NET class of same name</td>
</tr>
<tr>
<td>instances serialized by a PdxSerializer</td>
<td>.NET class of same name</td>
</tr>
<tr>
<td>java.lang.Byte</td>
<td>System.SByte</td>
</tr>
<tr>
<td>java.lang.Boolean</td>
<td>System.Boolean</td>
</tr>
<tr>
<td>java.lang.Character</td>
<td>System.Char</td>
</tr>
<tr>
<td>java.lang.Short</td>
<td>System.Int16</td>
</tr>
<tr>
<td>Java Type</td>
<td>.NET Type</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>java.lang.Integer</td>
<td>System.Int32</td>
</tr>
<tr>
<td>java.lang.Long</td>
<td>System.Int64</td>
</tr>
<tr>
<td>java.lang.Float</td>
<td>System.Float</td>
</tr>
<tr>
<td>java.lang.Double</td>
<td>System.Double</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>System.String</td>
</tr>
<tr>
<td>java.util.Date</td>
<td>System.DateTime</td>
</tr>
<tr>
<td>byte[]</td>
<td>System.Byte[]</td>
</tr>
<tr>
<td>boolean[]</td>
<td>System.Boolean[]</td>
</tr>
<tr>
<td>char[]</td>
<td>System.Char[]</td>
</tr>
<tr>
<td>short[]</td>
<td>System.Int16[]</td>
</tr>
<tr>
<td>int[]</td>
<td>System.Int32[]</td>
</tr>
<tr>
<td>long[]</td>
<td>System.Int64[]</td>
</tr>
<tr>
<td>float[]</td>
<td>System.Float[]</td>
</tr>
<tr>
<td>double[]</td>
<td>System.Double[]</td>
</tr>
<tr>
<td>String[]</td>
<td>System.String[]</td>
</tr>
<tr>
<td>byte[][]</td>
<td>System.Byte[][]</td>
</tr>
<tr>
<td>Object[]</td>
<td>System.Collections.Generic.List&lt;Object&gt;</td>
</tr>
<tr>
<td>java.util.Hashtable</td>
<td>System.Collections.Hashtable</td>
</tr>
<tr>
<td>java.util.ArrayList</td>
<td>System.Collections.Generic.IList&lt;Object&gt;</td>
</tr>
<tr>
<td>java.util.Vector</td>
<td>Collections.ArrayList</td>
</tr>
<tr>
<td>java.util.HashSet</td>
<td>CacheableHashSet</td>
</tr>
<tr>
<td>java.util.LinkedHashSet</td>
<td>CacheableLinkedHashSet</td>
</tr>
</tbody>
</table>
System Statistics

System Statistics provides information on the Pivotal GemFire installation and includes standard statistics for caching and distribution activities.

Statistics that end with "time" are time-based statistics. For performance reasons, by default, the system does not collect these. To enable time-based statistics gathering, set the system property `enable-time-statistics` in Table 9: Attributes in gfcpp.properties: Statistics Archiving Properties.

Sampling Statistics

When applications and cache servers join a distributed system, they indicate whether to enable statistics sampling and whether to archive the statistics that are gathered.

The following statistics are related to the statistic sampler.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sampleCount</td>
<td>Total number of samples taken by this sampler.</td>
</tr>
<tr>
<td>sampleTime</td>
<td>Total amount of time spent taking samples.</td>
</tr>
<tr>
<td>StatSampler</td>
<td>Statistics on the statistic sampler.</td>
</tr>
</tbody>
</table>

For more information about configuring statistics, see Attributes in gfcpp.properties.

System Performance Statistics

Performance statistics are collected for each application or cache server that connects to a distributed system.
## Region Statistics

These methods help you to get the statistics of a region.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>creates</td>
<td>Total number of cache creates for this region.</td>
</tr>
<tr>
<td>puts</td>
<td>Total number of cache put operations for this region.</td>
</tr>
<tr>
<td>putTime</td>
<td>Total time spent doing put operations for this region.</td>
</tr>
<tr>
<td>putAll</td>
<td>Total number of cache putAlls for this region.</td>
</tr>
<tr>
<td>putAllTime</td>
<td>Total time spent doing putAll operations for this region.</td>
</tr>
<tr>
<td>gets</td>
<td>Total number of cache gets for this region.</td>
</tr>
<tr>
<td>getTime</td>
<td>Total time spent doing get operations for this region.</td>
</tr>
<tr>
<td>getAll</td>
<td>Total number of cache getAlls for this region.</td>
</tr>
<tr>
<td>getAllTime</td>
<td>Total time spent doing getAll operations for this region.</td>
</tr>
<tr>
<td>hits</td>
<td>Total number of cache hits for this region.</td>
</tr>
<tr>
<td>misses</td>
<td>Total number of cache misses for this region.</td>
</tr>
<tr>
<td>entries</td>
<td>Current number of cache entries for this region.</td>
</tr>
<tr>
<td>destroys</td>
<td>Total number of cache destroys for this region.</td>
</tr>
<tr>
<td>clears</td>
<td>Total number of cache clears for this region.</td>
</tr>
<tr>
<td>overflows</td>
<td>Total number of cache overflows to disk for this region.</td>
</tr>
<tr>
<td>retrieves</td>
<td>Total number of cache entries fetched from disk into the cache region.</td>
</tr>
<tr>
<td>nonSingleHopCount</td>
<td>Total number of times client request required multiple hops.</td>
</tr>
<tr>
<td>metaDataRefreshCount</td>
<td>Total number of times metadata was refreshed due to the observation of multiple hops.</td>
</tr>
<tr>
<td>cacheLoaderCallCompleted</td>
<td>Total number of times a load has completed for this region.</td>
</tr>
<tr>
<td>cacheLoaderCallTime</td>
<td>Total time spent invoking the loaders for this region.</td>
</tr>
<tr>
<td>CacheWriterCallsCompleted</td>
<td>Total number of times a cache writer call has completed for this region.</td>
</tr>
<tr>
<td>CacheWriterCallTime</td>
<td>Total time spent doing cache writer calls.</td>
</tr>
<tr>
<td>CacheListenerCalls Completed</td>
<td>Total number of times a cache listener call has completed for this region.</td>
</tr>
<tr>
<td>CacheListenerCallTime</td>
<td>Total time spent doing cache listener calls for this region.</td>
</tr>
</tbody>
</table>
Cache Performance Statistics

Use cache performance statistics to determine the type and number of cache operations being performed and how much time they consume.

These statistics are available if the member creates a cache.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>creates</td>
<td>Total number of cache creates.</td>
</tr>
<tr>
<td>puts</td>
<td>Total number of cache puts.</td>
</tr>
<tr>
<td>gets</td>
<td>Total number of cache gets.</td>
</tr>
<tr>
<td>entries</td>
<td>Current number of cache entries.</td>
</tr>
<tr>
<td>hits</td>
<td>Total number of cache hits.</td>
</tr>
<tr>
<td>misses</td>
<td>Total number of cache misses.</td>
</tr>
<tr>
<td>destroys</td>
<td>Total number of cache destroys.</td>
</tr>
<tr>
<td>overflows</td>
<td>Total number of cache overflows to persistence backup.</td>
</tr>
<tr>
<td>cacheListenerCallsCompleted</td>
<td>Total number of times a cache listener call has completed.</td>
</tr>
<tr>
<td>pdxInstanceDeserializations</td>
<td>Total number of times getObject has been called on a PdxInstance.</td>
</tr>
<tr>
<td>pdxInstanceDeserializationTime</td>
<td>Total amount of time, in nanoseconds, spent deserializing PdxInstances by calling getObject.</td>
</tr>
<tr>
<td>pdxInstanceCreations</td>
<td>Total number of times a deserialization created a PdxInstance.</td>
</tr>
<tr>
<td>pdxSerializations</td>
<td>Total number of PDX serializations.</td>
</tr>
<tr>
<td>pdxSerializedBytes</td>
<td>Total number of bytes produced by PDX serialization.</td>
</tr>
<tr>
<td>pdxDeserializations</td>
<td>Total number of PDX deserializations.</td>
</tr>
<tr>
<td>pdxDeserializedBytes</td>
<td>Total number of bytes read by PDX deserialization.</td>
</tr>
<tr>
<td>tombstoneCount</td>
<td>Total number of tombstone entries created for performing concurrency checks.</td>
</tr>
<tr>
<td>nonReplicatedTombstoneSize</td>
<td>Approximate total size (in bytes) of tombstones present in the client cache.</td>
</tr>
</tbody>
</table>
Continuous Query Statistics

Continuous query statistics give you information about a registered Continuous Query (CQ) represented by the CqQuery object.

<table>
<thead>
<tr>
<th>inserts</th>
<th>Total number of inserts for this CQ query.</th>
</tr>
</thead>
<tbody>
<tr>
<td>updates</td>
<td>Total number of updates for this CQ query.</td>
</tr>
<tr>
<td>deletes</td>
<td>Total number of deletes for this CQ query.</td>
</tr>
<tr>
<td>events</td>
<td>Total number of events for this CQ query.</td>
</tr>
</tbody>
</table>
## CQ Service Statistics

Use CQ service methods to get aggregate statistical information about the continuous queries of a client.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CqsActive</td>
<td>Total number of CQs active for this CQ service.</td>
</tr>
<tr>
<td>CqsCreated</td>
<td>Total number of CQs created for this CQ service.</td>
</tr>
<tr>
<td>CqsClosed</td>
<td>Total number of CQs closed for this CQ service.</td>
</tr>
<tr>
<td>CqsStopped</td>
<td>Total number of CQs stopped for this CQ service.</td>
</tr>
<tr>
<td>CqsOnClient</td>
<td>Total number of CQs on the client for this CQ service.</td>
</tr>
</tbody>
</table>
## Pool Statistics

Use the pool object to get statistics on connection pools.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>locators</td>
<td>Current number of locators discovered.</td>
</tr>
<tr>
<td>servers</td>
<td>Current number of servers discovered.</td>
</tr>
<tr>
<td>locatorRequests</td>
<td>Number of requests from this connection pool to a locator.</td>
</tr>
<tr>
<td>locatorResponses</td>
<td>Number of responses from the locator to this connection pool.</td>
</tr>
<tr>
<td>poolConnections</td>
<td>Current number of pool connections.</td>
</tr>
<tr>
<td>connects</td>
<td>Total number of times a connection has been created.</td>
</tr>
<tr>
<td>ConnectionWaitTime</td>
<td>Total time (nanoseconds) spent waiting for a connection.</td>
</tr>
<tr>
<td>disconnects</td>
<td>Total number of times a connection has been destroyed.</td>
</tr>
<tr>
<td>minPoolSizeConnect</td>
<td>Total number of connects done to maintain minimum pool size.</td>
</tr>
<tr>
<td>loadConditioningConnects</td>
<td>Total number of connects done due to load conditioning.</td>
</tr>
<tr>
<td>idleDisconnects</td>
<td>Total number of disconnects done due to idle expiration.</td>
</tr>
<tr>
<td>loadConditioningDisconnects</td>
<td>Total number of disconnects done due to load conditioning expiration.</td>
</tr>
<tr>
<td>connectionWaitsInProgress</td>
<td>Current number of threads waiting for a connection.</td>
</tr>
<tr>
<td>connectionWaits</td>
<td>Total number of times a thread completed waiting for a connection (by timing out or by getting a connection).</td>
</tr>
<tr>
<td>clientOpsInProgress</td>
<td>Current number of clientOps being executed.</td>
</tr>
<tr>
<td>clientOps</td>
<td>Total number of clientOps completed successfully.</td>
</tr>
<tr>
<td>clientOpFailures</td>
<td>Total number of clientOp attempts that have failed.</td>
</tr>
<tr>
<td>clientOpTimeouts</td>
<td>Total number of clientOp attempts that have timed out.</td>
</tr>
<tr>
<td>QueryExecutions</td>
<td>Total number of queryExecutions.</td>
</tr>
<tr>
<td>QueryExecutionTime</td>
<td>Total time spent while processing queryExecution.</td>
</tr>
</tbody>
</table>
### Delta Statistics

Delta statistics provide information about updates to data.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deltaMessageFailures</td>
<td>Total number of messages containing delta (received from server) but could not be processed after reception.</td>
</tr>
<tr>
<td>deltaPuts</td>
<td>Total number of puts containing delta that have been sent from client to server.</td>
</tr>
<tr>
<td>processedDeltaMessages</td>
<td>Total number of messages containing delta received from server and processed after reception.</td>
</tr>
<tr>
<td>processedDeltaMessagesTime</td>
<td>Total time spent applying delta (received from server) on existing values at client.</td>
</tr>
</tbody>
</table>

### Operating System Statistics

Use operating system statistics to determine a member's CPU, memory, and disk usage.
# Linux Process Statistics

Use these methods to get information about a Linux operating system process that is using a GemFire system.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imageSize</td>
<td>The size of the process's image in megabytes.</td>
</tr>
<tr>
<td>rssSize</td>
<td>The size of the process's resident set size in megabytes.</td>
</tr>
<tr>
<td>userTime</td>
<td>The operating system statistic for the process CPU usage in user time</td>
</tr>
<tr>
<td>systemTime</td>
<td>The operating system statistic for the process CPU usage in system time.</td>
</tr>
<tr>
<td>hostCpuUsage</td>
<td>The operating system statistic for the host CPU usage.</td>
</tr>
<tr>
<td>threads</td>
<td>Number of threads currently active in this process.</td>
</tr>
<tr>
<td>LinuxProcessStats</td>
<td>Statistics for a Linux process.</td>
</tr>
</tbody>
</table>
### Solaris Process Statistics

Use these methods to get information about a Solaris operating system process that is using a GemFire system.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imageSize</td>
<td>Size of the process image in megabytes.</td>
</tr>
<tr>
<td>rssSize</td>
<td>Size of the process resident set in megabytes.</td>
</tr>
<tr>
<td>userTime</td>
<td>Operating system statistic for the process CPU usage in user time.</td>
</tr>
<tr>
<td>systemTime</td>
<td>Operating system statistic for the process CPU usage in system time.</td>
</tr>
<tr>
<td>processCpuUsage</td>
<td>Operating system statistic for the CPU usage of this process.</td>
</tr>
<tr>
<td>hostCpuUsage</td>
<td>Operating system statistic for the host CPU usage.</td>
</tr>
<tr>
<td>threads</td>
<td>Number of threads currently active in this process.</td>
</tr>
</tbody>
</table>
# Windows Process Statistics

Use these methods to get information about a Windows operating system process that is using a GemFire system.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handles</td>
<td>Total number of handles currently open by this process. This number is the sum of the handles currently open by each thread in this process.</td>
</tr>
<tr>
<td>priorityBase</td>
<td>Current base priority of the process. Threads within a process can raise and lower their own base priority relative to the process's base priority.</td>
</tr>
<tr>
<td>threads</td>
<td>Number of threads currently active in this process. An instruction is the basic unit of execution in a processor, and a thread is the object that executes instructions. Every running process has at least one thread.</td>
</tr>
<tr>
<td>activeTime</td>
<td>Elapsed time in milliseconds that all threads of this process used the processor to execute instructions. Code executed to handle some hardware interrupts and trap conditions are included in this count.</td>
</tr>
<tr>
<td>pageFaults</td>
<td>Total number of Page Faults by the threads executing in this process. A page fault occurs when a thread refers to a virtual memory page that is not in its working set in main memory. This will not cause the page to be fetched from disk if it is on the standby list and hence already in main memory, or if it is in use by another process with whom the page is shared.</td>
</tr>
<tr>
<td>pageFileSize</td>
<td>Current number of bytes this process has used in the paging file(s). Paging files are used to store pages of memory used by the process that are not contained in other files. Paging files are shared by all processes, and lack of space in paging files can prevent other processes from allocating memory.</td>
</tr>
<tr>
<td>pageFileSizePeak</td>
<td>Maximum number of bytes this process has used in the paging file(s). Paging files are used to store pages of memory used by the process that are not contained in other files. Paging files are shared by all processes, and lack of space in paging files can prevent other processes from allocating memory.</td>
</tr>
<tr>
<td>privateSize</td>
<td>Current number of bytes this process has allocated that cannot be shared with other processes.</td>
</tr>
<tr>
<td>Metric</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>systemTime</td>
<td>Elapsed time in milliseconds that the threads of the process have spent executing code in privileged mode. When a Windows system service is called, the service often runs in Privileged Mode to gain access to system-private data. Such data is protected from access by threads executing in user mode. Calls to the system can be explicit or implicit, such as page faults or interrupts. Unlike some early operating systems, Windows uses process boundaries for subsystem protection in addition to the traditional protection of user and privileged modes. These subsystem processes provide additional protection. Therefore, some work done by Windows on behalf of your application might appear in other subsystem processes in addition to the privileged time in your process.</td>
</tr>
<tr>
<td>userTime</td>
<td>Elapsed time in milliseconds that this process's threads have spent executing code in user mode. Applications, environment subsystems, and integral subsystems execute in user mode. Code executing in User Mode cannot damage the integrity of the Windows Executive, Kernel, and device drivers. Unlike some early operating systems, Windows uses process boundaries for subsystem protection in addition to the traditional protection of user and privileged modes. These subsystem processes provide additional protection. Therefore, some work done by Windows on behalf of your application might appear in other subsystem processes in addition to the privileged time in your process.</td>
</tr>
<tr>
<td>activeTime</td>
<td>Elapsed time in milliseconds that all threads of this process used the processor to execute instructions. An instruction is the basic unit of execution in a computer, a thread is the object that executes instructions, and a process is the object created when a program is run. Code executed to handle some hardware interrupts and trap conditions are included in this count.</td>
</tr>
<tr>
<td>pageFaults</td>
<td>Total number of page faults by the threads executing in this process. A page fault occurs when a thread refers to a virtual memory page that is not in its working set in main memory. This will not cause the page to be fetched from disk if it is on the standby list and hence already in main memory, or if it is in use by another process with whom the page is shared.</td>
</tr>
<tr>
<td><strong>Page</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>pageFileSize</td>
<td>Current number of bytes this process has used in the paging file(s). Paging files are used to store pages of memory used by the process that are not contained in other files. Paging files are shared by all processes, and lack of space in paging files can prevent other processes from allocating memory.</td>
</tr>
<tr>
<td>pageFileSizePeak</td>
<td>Maximum number of bytes this process has used in the paging file(s). Paging files are used to store pages of memory used by the process that are not contained in other files. Paging files are shared by all processes, and lack of space in paging files can prevent other processes from allocating memory.</td>
</tr>
<tr>
<td>privateSize</td>
<td>Current number of bytes this process has allocated that cannot be shared with other processes.</td>
</tr>
<tr>
<td>systemTime</td>
<td>Elapsed time in milliseconds that the threads of the process have spent executing code in privileged mode. When a Windows system service is called, the service often runs in privileged mode to gain access to system-private data. Such data is protected from access by threads executing in user mode. Calls to the system can be explicit or implicit, such as page faults or interrupts. Unlike some early operating systems, Windows uses process boundaries for subsystem protection in addition to the traditional protection of user and privileged modes. These subsystem processes provide additional protection. Therefore, some work done by Windows on behalf of your application might appear in other subsystem processes in addition to the privileged time in your process.</td>
</tr>
<tr>
<td>userTime</td>
<td>Elapsed time in milliseconds that this process's threads have spent executing code in user mode. Applications, environment subsystems, and integral subsystems execute in user mode. Code executing in User Mode cannot damage the integrity of the Windows Executive, Kernel, and device drivers. Unlike some early operating systems, Windows uses process boundaries for subsystem protection in addition to the traditional protection of user and privileged modes. These subsystem processes provide additional protection. Therefore, some work done by Windows on behalf of your application might appear in other subsystem processes in addition to the privileged time in your process.</td>
</tr>
<tr>
<td><strong>virtualSize</strong></td>
<td>Current size in bytes of the virtual address space the process is using. Use of virtual address space does not necessarily imply corresponding use of either disk or main memory pages. Virtual space is finite, and by using too much, the process can limit its ability to load libraries.</td>
</tr>
<tr>
<td><strong>virtualSizePeak</strong></td>
<td>Maximum number of bytes of virtual address space the process has used at any one time. Use of virtual address space does not necessarily imply corresponding use of either disk or main memory pages. Virtual space is however finite, and by using too much, the process might limit its ability to load libraries.</td>
</tr>
<tr>
<td><strong>workingSetSize</strong></td>
<td>Current number of bytes in the Working Set of this process. The Working Set is the set of memory pages touched recently by the threads in the process. If free memory in the computer is above a threshold, pages are left in the Working Set of a process even if they are not in use. When free memory falls below a threshold, pages are trimmed from Working Sets. If pages are needed, they are then soft-faulted back into the Working Set before they are paged out to disk.</td>
</tr>
<tr>
<td><strong>workingSetSizePeak</strong></td>
<td>Maximum number of bytes in the Working Set of this process at any point in time. The Working Set is the set of memory pages touched recently by the threads in the process. If free memory in the computer is above a threshold, pages are left in the Working Set of a process even if they are not in use. When free memory falls below a threshold, pages are trimmed from Working Sets. If they are needed they will then be soft faulted back into the Working Set before they leave main memory.</td>
</tr>
<tr>
<td><strong>cpuUsage</strong></td>
<td>Percentage CPU used by this process.</td>
</tr>
<tr>
<td><strong>WindowsProcessStats</strong></td>
<td>Statistics for a Microsoft Windows process.</td>
</tr>
</tbody>
</table>
Part XXI

Installing the SQLite Persistence Manager

*Installing the SQLite Persistence Manager* describes how to download, build and install the SQLite database libraries for use with disk overflow.

The commands that you type are shown in boldface fixed font. See *PersistenceManager* for additional information about the SQLite database libraries.

**Linux Installation**

This topic describes how to install the SQLite Persistence Manager on Linux for use with the GemFire native client.

The `productDir` directory refers to the path to the native client product directory.

The following libraries must be present in the runtime linking path:

- `libSqLiteImpl.so` is provided in `productDir/lib`, so it is already present in the runtime linking path.
- `libsqlite3.so` is the SQLite Library. You need to create this library and make available in the runtime linking path, or copied to `productDir/lib`, as described below.

Pivotal GemFire Native Client has been tested with SQLite version 3.7.14.1.

**Downloading, Building and Installing the Library**

You create the SQLite database library by downloading the latest .zip file and compiling the source code.

1. Download the source code `sqlite-autoconf-NNNNNNN.tar.gz` file (where `NNNNNNN` corresponds to the version) for SQLite version 3.7.14.1 or later from [http://www.sqlite.org/download.html](http://www.sqlite.org/download.html).
2. Extract the source code from the .tar.gz file. For example:

   ```
   tar -xvf sqlite-autoconf-3071401.tar.gz
   ```

3. Change directories to the extracted source files, and follow the install instructions located in the "INSTALL" file.

   a. Run the `configure` command for 32-bit or 64-bit with the following options, all entered on a single command line. Change the `--prefix` directory specification to the location where you want the libraries:

      - **32-bit:**

        ```
        CFLAGS="-m32" ./configure --prefix=/desired_binary_location/sqlite-binaries
        ```

      - **64-bit:**

        ```
        ./configure --prefix=/desired_binary_location/sqlite-binaries
        ```

   b. Run `gmake install` as described in the build instructions. The libraries will be available in the `sqlite-binaries` directory that you specified.
Solaris Installation

This topic describes how to install the SQLite Persistence Manager on Solaris for use with the GemFire native client.

The `productDir` directory refers to the path to the native client product directory.

The following libraries must be present in the runtime linking path:

- `libSqLiteImpl.so` is provided in `productDir/lib`, so it is already present in the runtime linking path.
- `libsqlite3.so` is the SQLite Library. You need to create this library and make available in the runtime linking path, or copied to `productDir/lib`, as described below.

Pivotal GemFire Native Client has been tested with SQLite version 3.7.14.1.

Downloading, Building, and Installing the Library

You create the SQLite database library by downloading the latest .zip file and compiling the source code.

1. Download the source code `sqlite-autoconf-NNNNNNN.tar.gz` file (where `NNNNNNN` corresponds to the version) for SQLite version 3.7.14.1 or later from http://www.sqlite.org/download.html.
2. Update your PATH environment variable to include the location of the Solaris `ar` command.
   ```
   export PATH=/usr/css/bin:$PATH
   ```
3. Extract the source code from the .tar.gz file. First unzip:
   ```
   gzip -d sqlite-autoconf-3071401.tar.gz
   ```
   Then untar the file:
   ```
   tar -xvf sqlite-autoconf-3071401.tar
   ```
4. Change directories to the extracted source files, and follow the install instructions located in the "INSTALL" file.
   a. Run the configure command for 32-bit or 64-bit Solaris systems with the following options, all entered on a single command line. Change the `--prefix` directory specification to the location where you want the libraries:
      - 32-bit:
        ```
        CC=cc CFLAGS="-xarch=v8plus -code=pic32" ./configure --prefix=/desired_binary_location/sqlite-binaries
        ```
      - 64-bit:
        ```
        CC=cc CFLAGS="-xarch=v9 -code=pic32" ./configure --prefix=/desired_binary_location/sqlite-binaries CFLAGS="-m64"
        ```
   b. Run `gmake install`. The libraries will be available in the `sqlite-binaries` directory that you specified.

Windows Installation

This topic describes how to install the SQLite Persistence Manager on Windows for use with the Pivotal GemFire native client.

Pivotal GemFire Native Client has been tested with SQLite version 3.7.14.1.

The `productDir` directory refers to the native client product directory path.

The following libraries are required. The `productDir\bin` directory containing these libraries must be present in the Windows PATH environment variable, and that directory is added to PATH during the Pivotal GemFire product installation.

- The `sqliteimpl.dll` and `GemStone.GemFire.Plugins.SQLite.dll` files are provided in `productDir\bin`.
- For .NET C# native client application development, you need to obtain the `System.Data.SQLite.dll` SQLite library, as described below. The library can be copied to `productDir\bin`.
- For C++ native client application development, you need the `Sqlite3.dll` SQLite Library. You create this library and make it available in the runtime linking path, or copied to `productDir\bin`, as described below.

Downloading Pre-built System.Data.SQLite.dll Binaries

If you are writing native client applications using the .NET caching API, obtain the SQLite library (version 3.7.14.1 or later) for Windows as follows:

2. Download the appropriate setup file for your .NET Framework installation and hardware architecture.
   - For 32-bit Windows, under Setups for 32-bit Windows (.NET Framework 4.0) download `sqlite-netFx40-setup-x86-2010-1.0.84.0.exe`.
   - For 64-bit Windows, under Setups for 64-bit Windows (.NET Framework 4.0) download `sqlite-netFx40-setup-x64-2010-1.0.84.0.exe`.
3. Execute the setup .exe file, and follow the prompts in the installation wizard. Accept all default installation options.
4. Copy the `C:\Program Files\System.Data.SQLite\2010\bin\System.Data.SQLite.dll` file to your Pivotal GemFire Native Client distribution at `productDir\bin`.

Downloading, Building, and Installing the Library

If you are writing native client applications using the C++ caching API, you need to build the SQLite solution for your Windows platform architecture.

1. Download the source code `sqlite-autoconf-NNNNNNNN.tar.gz` file (where NNNNNNNN corresponds to the version) for SQLite version 3.7.14.1 or later from `http://www.sqlite.org/download.html`.
2. Extract the source code from the .tar.gz file. You may need to use CygWin or a Windows-compatible tar extraction tool.
3. Using Visual Studio 2010, build the SQLite_3.7.14.1 solution either as a release or debug build:
   - If you are using 32-bit Windows, build the SQLite_3.7.14.1, use the win32 configuration.
   - If you are using 64-bit Windows, build the SQLite_3.7.14.1, use the x64 configuration.
4. From the built files, copy the `Sqlite3.dll` file to your Pivotal GemFire Native Client distribution at `productDir\bin`. 
Glossary

This glossary defines terms used in Pivotal GemFire documentation.

**API**
Application Programming Interface. GemFire provides APIs to cached data for C++ and .NET applications.

**application program**
A program designed to perform a specific function directly for the user or, in some cases, for another application program. GemFire applications use the GemFire application programming interfaces (APIs) to modify cached data.

**cache**
A cache created by an application or cache server process. For the process, its cache is the point of access to all caching features and the only view of the cache that is available. Cache creation requires membership in the distributed system. See also local cache and remote cache.

**cache configuration file**
An XML file that declares the initial configuration of a cache, commonly named cache.xml. C++ and .NET applications can configure the cache additionally through the GemFire programming APIs.

**cache listener**
User-implemented plug-in for receiving and handling region entry events. A region's cache listener is called after an entry in the local cache is modified.

**cache loader**
User-implemented plug-in for loading data into a region. A region's cache loader is used to load data that is requested of the region but is not available in the distributed system. For a distributed region, the loader that is used can be in a different cache from the one where the data-request operation originated. See also cache writer and netSearch.

**cache server**
A long-running, configurable caching process, generally used to serve cached data to the applications. Usually, cache servers are configured to operate as servers in a client-server typology and their regions are configured to be replicates. See also server.

**cache writer**
User-implemented plug-in intended for synchronizing the cache with an outside data source. A region's cache writer is a synchronous listener to cache data events. The cache writer has the ability to abort a data modification. See also cache loader.
caching enabled
Specifies whether data is cached in the region. GemFire gives you the option of running applications without entry caching. For example, you can configure a distributed system as a simple messaging service.

client
In a client-server topology, clients can connect to cache servers, create new regions on the cache server, and store data in the cache server region. Clients can also connect to existing regions on a cache server and do directed gets and puts on the cache server. Clients do not track membership information about other clients, nor do they share information with other clients.

concurrency level
An estimate of the number of threads expected to concurrently modify values in the region. The actual concurrency may vary; this value is used to optimize the allocation of system resources.

connection
What an application uses to access a GemFire distributed system. An application can connect to a GemFire system by calling the `DistributedSystem::connect` function with the appropriate parameter settings. An application must connect to a distributed system to gain access to GemFire functionality.

destroy
Remove an entry from a region or remove a region from a cache.

disk policy
Determines whether LRU entries exceeding the entries limit for a caching region are destroyed or written to disk.

distributed scope
Enables a region to automatically send entry value updates to remote caches and incorporate updates received from remote caches. The scope identifies whether distribution operations must wait for acknowledgement from other caches before continuing. A distributed region's `cache loader` and `cache writer` (defined in the local cache) can be invoked for operations originating in remote caches.

distributed system
One or more GemFire system members that have been configured to communicate with each other, forming a single, logical system. Also used for the object that is instantiated to create the connection between the distributed system members.

DTD
Document Type Definition. A language that describes the contents of a Standard Generalized Markup Language (SGML) document. The DTD is also used with XML. The DTD definitions can be embedded within an XML document or in a separate file.

entry
A data object in a region. A region entry consists of a key and a value. The value is either null (invalid) or an object. A region entry knows what region it is in. See also `region data`, `entry key`, and `entry value`.

entry key
The unique identifier for an entry in a region.
**entry value**
The data contained in an entry.

**expiration**
A cached object expires when its time-to-live or idle timeout counters are exhausted. A region has one set of expiration attributes for itself and one set for all region entries.

**expiration action**
The action to be taken when a cached object expires. The expiration action specifies whether the object is to be invalidated or destroyed, and whether the action is to be performed only in the local cache or throughout the distributed system. A destroyed object is completely removed from the cache. A region is invalidated by invalidating all entries contained in the region. An entry is invalidated by having its value marked as invalid.

Expiration attributes are set at the region level for the region and at the entry level for entries. See also idle timeout and time-to-live.

**factory method**
An interface for creating an object which at creation time can let its subclasses decide which class to instantiate. The factory method helps instantiate the appropriate subclass by creating the correct object from a group of related classes.

**idle timeout**
The amount of time a region or region entry may remain in the cache unaccessed before being expired. Access to an entry includes any get operation and any operation that resets the entry's time-to-live counter. Region access includes any operation that resets an entry idle timeout, and any operation that resets the region's time-to-live.

Idle timeout attributes are set at the region level for the region and at the entry level for entries. See also time-to-live and expiration action.

**interest list**
A mechanism that allows a region to maintain information about receivers for a particular key-value pair in the region, and send out updates only to those nodes. Interest lists are particularly useful when you expect a large number of updates on a key as part of the entry life cycle.

**invalid**
The state of an object when the cache holding it does not have the current value of the object.

**invalidate**
Remove only the value of an entry in a cache, not the entry itself.

**listener**
An event handler. The listener registers its interest in one or more events and is notified when the events occur.

**load factor**
A region attribute that sets initial parameters on the underlying hashmap used for storing region entries.
**local cache**
The part of the distributed cache that is resident in the current process. This term is used to differentiate the cache where a specific operation is being performed from other caches in the distributed system. See also remote cache.

**local scope**
Enables a region to hold a private data set that is not visible to other caches. See also scope.

**LRU**
Least Recently Used. Refers to a region entry or entries most eligible for eviction due to lack of interest by client applications.

**LRU entries limit**
A region attribute that sets the maximum number of entries to hold in a caching region. When the capacity of the caching region is exceeded, LRU is used to evict entries.

**membership**
Applications and cache servers connect to a GemFire distributed system by invoking the static function DistributedSystem::connect. Through this connection, the application gains access to the APIs for distributed data caches. When a C++ or .NET application connects to a distributed system, it specifies the system it is connecting to by indicating the communication protocol and address to use to find other system members.

**netSearch**
The method used by GemFire to search remote caches for a data entry that is not found in the local cache region. This operates only on distributed regions.

**overflows**
An eviction option that causes the values of LRU entries to be moved to disk when the region reaches capacity. See disk policy.

**persistence manager**
The persistence manager manages the memory-to-disk and disk-to-memory actions for LRU entries. See overflows.

**region**
A logical grouping of data within a cache. Regions are used to store data entries (see entry). Each region has a set of attributes governing activities such as expiration, distribution, data loading, events, and eviction control.

**region attributes**
The class of attributes governing the creation, location, distribution, and management of a region and its entries.

**region data**
All of the entries directly contained in the region.

**region entry**
See entry.
remote cache
Any part of the distributed cache that is resident in a process other than the current one. If an application or cache server does not have a data entry in the region in its local cache, it can do a netSearch in an attempt to retrieve the entry from the region in a remote cache. See also local cache.

scope
Region attribute. Identifies whether a region keeps its entries private or automatically sends entry value updates to remote caches and incorporates updates received from remote caches. The scope also identifies whether distribution operations must wait for acknowledgement from other caches before continuing. See also distributed scope and local scope.

serialization
The process of converting an object graph to a stream of bytes.

server
In a client-server topology, the server manages membership and allows remote operations. The server maintains membership information for its clients in the distributed system, along with information about peer applications and other servers in the system. See also cache server.

system member
A process that has established a connection to a distributed system.

time-to-live
The amount of time a region or region entry may remain in the cache without being modified before being expired. Entry modification includes creation, update, and removal. Region modification includes creation, update, or removal of the region or any of its entries.

Time-to-live attributes are set at the region level for the region, and at the entry level for entries. See also idle timeout and expiration action.

XML
EXtensible Markup Language. An open standard for describing data from the W3C, XML is a markup language similar to HTML. Both are designed to describe and transform data, but where HTML uses predefined tags, XML allows tags to be defined inside the XML document itself. Using XML, virtually any data item can be identified. The XML programmer creates and implements data-appropriate tags whose syntax is defined in a DTD file or an XSD (XML schema definition.)

XML schema definition
The definition of the structure, content, and semantics used in an XML document. The definition can be used to verify that each item of content in a document adheres to the specification of the element in which the content is placed. The XML schema is a superset of DTD. Unlike DTD, XML schemas are written in XML syntax, which, although more verbose than DTD, are more descriptive and can have stronger typing. Files containing XML schema definitions generally have the XSD extension.

XSD
See XML schema definition.