

The GemFire logo consists of the word "GEMFIRE" in a bold, black, sans-serif font, with a small orange triangle above the letter "I". A thin green horizontal line is positioned below the word. Below "GEMFIRE" is the word "ENTERPRISE" in a smaller, black, sans-serif font. The background features a large, light gray diamond shape composed of smaller triangles.

**GEMFIRE**<sup>®</sup>  
ENTERPRISE

*Early Access Features*

Version 5.8 Limited Availability

November 2008

Send comments on this manual to [docs@gemstone.com](mailto:docs@gemstone.com)

---

## INTELLECTUAL PROPERTY OWNERSHIP

This documentation is furnished for informational use only and is subject to change without notice. GemStone Systems, Inc. assumes no responsibility or liability for any errors or inaccuracies that may appear in this documentation.

This documentation, or any part of it, may not be reproduced, displayed, photocopied, transmitted, or otherwise copied in any form or by any means now known or later developed, such as electronic, optical, or mechanical means, without express written authorization from GemStone Systems, Inc.

Warning: This computer program and its documentation are protected by copyright law and international treaties. Any unauthorized copying or distribution of this program, its documentation, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted under the maximum extent possible under the law.

The software installed in accordance with this documentation is copyrighted and licensed by GemStone Systems, Inc. under separate license agreement. This software may only be used pursuant to the terms and conditions of such license agreement. Any other use may be a violation of law.

Use, duplication, or disclosure by the Government is subject to restrictions set forth in the Commercial Software - Restricted Rights clause at 52.227-19 of the Federal Acquisitions Regulations (48 CFR 52.227-19) except that the government agency shall not have the right to disclose this software to support service contractors or their subcontractors without the prior written consent of GemStone Systems, Inc.

This software is provided by GemStone Systems, Inc. and contributors "as is" and any expressed or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall GemStone Systems, Inc. or any contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

## COPYRIGHTS

This software product, its documentation, and its user interface Copyright © 1997-2008, GemStone Systems, Inc. All Rights Reserved by GemStone Systems Inc.

Java Software technologies Copyright © 1994-2000 Sun Microsystems, Inc. All rights reserved.

Trove Log4J Copyright © 1999 The Apache Software Foundation. All rights reserved. The Trove library is licensed under the Lesser GNU Public License, which is included with the distribution in a file called LICENSE.txt. PrimeFinder and HashFunctions classes in Trove © Copyright 1999 CERN - European Organization for Nuclear Research.

JavaGroups copyright © 1999-2004 Free Software Foundation, Inc.

GNU Trove copyright © 2001-2004 Eric D. Friedman. The PrimeFinder and HashFunctions classes in Trove are copyright © 1999 CERN - European Organization for Nuclear Research. Copyright © 1991, 1999 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

MX4J project (<http://mx4j.sourceforge.net>). Copyright © 2001-2004 by the MX4J contributors. All rights reserved.

Commons Modeler Copyright © 2004 Commons Modeler. All rights reserved.

JDBM Copyright © 2000 Cees de Groot. All Rights Reserved. Contributions are Copyright © 2000 by their associated contributors.

Copyright © 1994 Hewlett-Packard Company

Copyright © 1996,97 Silicon Graphics Computer Systems, Inc. Copyright © 1997 Moscow Center for SPARC Technology.

Copyright © 1998-2003 Daniel Veillard. All rights reserved.

Jgroups © 2001, 2002 [www.jgroups.org](http://www.jgroups.org)

Antlr © 2005, Terence Parr. All rights reserved.

## PATENTS

GemFire is protected by U.S. patent 6,360,219. Additional patents pending.

## TRADEMARKS

GemStone, GemFire, GemFire Enterprise, and the GemStone logo are trademarks or registered trademarks of GemStone Systems, Inc. in the United States and other countries (trademark application pending for GemFire).

UNIX is a registered trademark of The Open Group in the U. S. and other countries.

Linux is a registered trademark of Linus Torvalds.

Red Hat and all Red Hat-based trademarks and logos are trademarks or registered trademarks of Red Hat, Inc. in the United States and other countries.

SUSE is a registered trademark of SUSE AG.

Sun, Sun Microsystems, Solaris, Forte, Java, Java Runtime Edition, JRE, and other Java-related marks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

Intel and Pentium are registered trademarks of Intel Corporation in the United States and other countries.

Microsoft, Windows, and Visual C++ are registered trademarks of Microsoft Corporation in the United States and other countries.

Exolab is a registered trademark of ExoLab Group.

IBM, AIX, and developerWorks are registered trademarks of IBM Corporation.

W3C is a registered trademark of the World Wide Web Consortium.

Other company or product names mentioned herein may be trademarks or registered trademarks of their respective owners. Trademark specifications are subject to change without notice. All terms mentioned in this documentation that are known to be trademarks or service marks have been appropriately capitalized to the best of our knowledge; however, GemStone cannot attest to the accuracy of all trademark information. Use of a term in this documentation should not be regarded as affecting the validity of any trademark or service mark.

## GemStone Systems, Inc.

1260 NW Waterhouse Avenue, Suite 200  
Beaverton, OR 97006

This document describes the early access features included in this release.

The functionality documented here is included as part of an early access program, with specific, targeted customers in mind. It may not yet be fully implemented, and it has not been subject to the full set of rigorous testing procedures that our General Availability software receives. If you wish to use this feature prior to its General Availability, we highly advise you to first inform your GemStone technical representative so additional support can be provided and any new requirements can be considered for implementation into the full release.

GemStone Systems, Inc. provides these features without warranty, and does not currently endorse their use in any critical applications.

For information on general access features provided with this release, refer to the *GemFire Enterprise Release Notes* for version 5.8 Limited Availability (LA). Refer to the programming guides and programming API documentation included with the product for information on using GemFire Enterprise to develop applications. For detailed information on administering GemFire Enterprise systems, see the *GemFire Enterprise System Administrator's Guide*. For details on programming with GemFire Enterprise, see the *GemFire Enterprise Developer's Guide*.

GemFire Enterprise 5.8 includes these early access features:

- ▶ [Function Execution - Early Access on page 4](#)
- ▶ [Eviction and Overflow in Partitioned Regions - Early Access on page 19](#)
- ▶ [Expiration in Partitioned Regions - Early Access on page 26](#)
- ▶ [Region Snapshots - Early Access on page 30](#)
- ▶ [EntryEvent Get Serialized Values - Early Access on page 30](#)
- ▶ [Bulk Operations Using getAll - Early Access on page 30](#)
- ▶ [Load Balancing in Multi-site Installations - Early Access on page 31](#)
- ▶ [Manual Start for Gateway Hubs - Early Access on page 36](#)
- ▶ [Write Behind Cache Listener Using Gateway Hubs- Early Access on page 37](#)

## Function Execution - Early Access

Using the GemFire Enterprise® function execution service, you can execute application functions on a single member, in parallel on a subset of members, or in parallel on all members of a distributed system. This section introduces the function execution service, and describes the various types of services. Implementation examples are also provided.

### Rationale

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 member or a small subset of 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 terms of data volume, where the number of data items managed rather than each item 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. The term we use to describe this routing of application code to the data of interest is aptly called *data-aware function routing*, or *behavior routing*.

### Function Execution Service

GemFire's new data-aware function execution service permits the execution of arbitrary application functions on the data fabric/grid members. These functions can be data-dependent functions or arbitrary functions that do not rely on any data that gets executed on the grid members. The next section provides more details on both types of functions and their uses.

#### Data-Dependent Functions

The function execution service permits the execution of arbitrary data-dependent application functions on the grid members - members where the data is striped for scale. The fundamental premise is to route the function transparently to the member that carries the data subset required by the application function to avoid moving data around on the network.

An application function can be executed on just one fabric member, executed in parallel on a subset of members, or in parallel across all members. This programming model is similar to the popular Map-Reduce model. Data-aware function routing is most appropriate for applications that require iteration over multiple data items (such a query or custom aggregation function). By colocating the relevant data and parallelizing the calculation, overall throughput can be dramatically increased. More importantly, the calculation latency is inversely proportional to the number of members on which it can be parallelized.

## Data-Independent Functions

Data-independent functions are executed on behalf of a client by targeting either a single server member, or executed in parallel on all the servers of a server group. The execution of a function on a single server member is similar to how applications execute stored procedures on database servers.

## Function Execution Details

Application functions can be targeted for execution on just a single member, in parallel on a subset of members that define a data region, or in parallel on all members that define a data region. Functions can be executed either synchronously (execute the function and wait for the result), or asynchronously (fire and forget).

Functions can be executed on replicated or partitioned regions, and they can be invoked from clients or peers in the GemFire cluster.

Data-dependent functions provide a hint indicating the data it is dependent on by providing region keys or routing keys (when using custom partitioned data region), or simply a data region name.

Data-independent functions can be executed on a specific server or a group of servers, on a specific member in the peer cluster, or on a group of members in the peer cluster.

The following code snippet shows the execution of a data-dependent function from a GemFire client.

### Example 1.1 Data-Aware Function Invoked from a Client

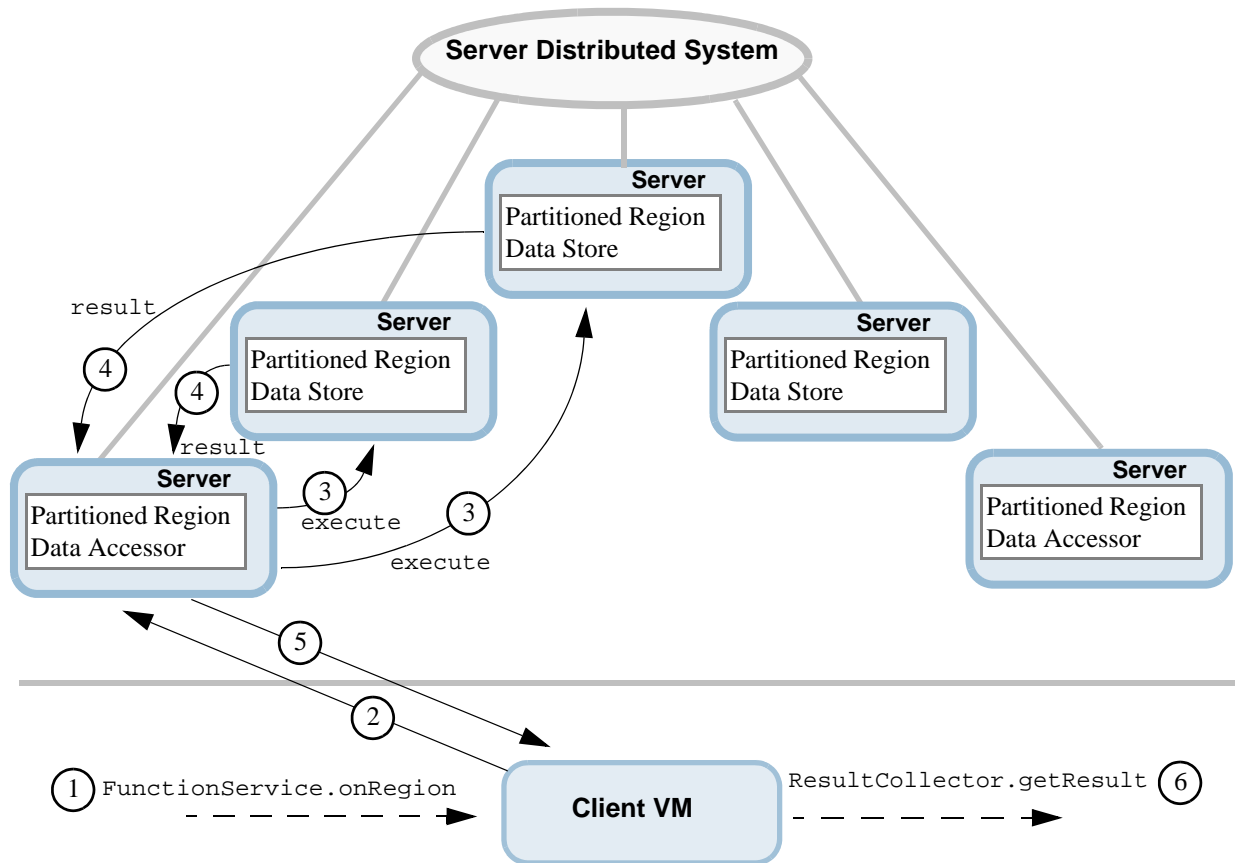
```
Region clientRegion ; // region has pool to communicate to server group
Set keySet = Collections.singleton("myKey");
Function multiGetFunction ;
Serializable args ;
ResultCollector rc = FunctionService.onRegion(clientRegion)
    .withArgs(args)
    .withFilter(keySet)
    .withCollector(new MyCustomResultCollector())
    .execute(multiGetFunction.getId());
// Application can do something else here before retrieving the result
Serializable functionResult = rc.getResult();
```

The previous code snippet assumes that the GemFire client VM has been started with a cache and has a client region that has been initialized with a pool that points to a running cache server that has the same region defined. Note that the data dependency is specified as the region with a keyset.

The `ResultCollector` interface in the previous example defines a container that gathers the results from function execution. `ResultCollector` provides these methods:

- ▶ `addResult`—Adds a single function execution result to the `ResultCollector`
- ▶ `endResults`—GemFire invokes this method when function execution has completed and all results for the execution have been obtained and added to the `ResultCollector`
- ▶ `getResult`—Returns the result of function execution, potentially blocking until all results are available
- ▶ `getResult(long timeout, com.gemstone.bp.edu.emory.mathcs.backport.java.util.concurrent.TimeUnit unit)`—Returns the result of function execution, blocking for the timeout period until all results are available

The following figure graphically illustrates the sequence of events that occur when a data-aware function is invoked from a client:

**Figure 1.1 Example of a Data-Aware Function Invoked from a Client****Example 1.2 Function Execution on Peers in a Distributed System**

```

Region replicatedRegion = cache.getRegion("myReplicatedRegion");
Set keySet = new HashSet();
keySet.add("key-1");
keySet.add("key-2");
Function multiGetFunction ;
Serializable args ;
ResultCollector rc =FunctionService.onRegion(replicatedRegion)
.withArgs(args)
.withFilter(keySet)
.withCollector(new MyCustomResultCollector())
.execute(multiGetFunction.getId());
// Application can do something else here
Serializable functionResult = rc.getResult();

```

The code snippet in the previous example shows a function being invoked on a replicated region. Replicated regions that act as data stores (as opposed to empty regions) can have function invoked on them. This snippet invokes `multiGetFunction` on all nodes that have defined `myReplicatedRegion`.

The previous example would work just as well if the region was a `PartitionedRegion`. When invoked on a partitioned region, the function executes on all nodes that have data stores defined for the region.

## Custom Partitioning and Data Colocation

In order to scale data in the GemFire cluster, data has to be partitioned across the cluster.

By default, GemFire uses a hashing policy where the key is hashed to compute a random bucket and is then mapped to a member to store the data. The physical location of the key-value pair is abstracted away from the application, so the application does not control the location of the data.

Custom partitioning allows applications to control colocation of related data entries. GemFire also allows the configuration of entry colocation across multiple partitioned regions.

For example, a financial risk analytical application wants to locate all trades, risk sensitivities, and reference data associated with a single instrument in the same region. Similarly, an order management system wants to locate all orders, line items, and shipments associated with a specific customer in one process space.

By collocating the specific data in the same region:

- ▶ The application can route a complex query to the member with the dataset required for the query and localize the entire query processing, increasing the speed of execution when compared to a distributed query.
- ▶ The applications that perform iterative operations on related datasets for aggregation can avoid unnecessary network hops. Compute-heavy applications that are also data-intensive can significantly increase the overall throughput of the application.

### Colocating Related Entries in a Single Partitioned Region

Applications implement the `PartitionResolver` interface to enable custom partitioning on a partitioned region. The following example describes the `PartitionResolver` API:

#### Example 1.3 Partition Resolver API for Colocating Data Entries on a Single Partitioned Region

```
public interface PartitionResolver extends CacheCallback, Declarable {
    public Serializable getRoutingObject(EntryOperation opDetails);
```

- ▶ `opDetails` represents the detail of the entry operation (`Region.get(Object)`)
- ▶ `getRoutingObject` returns the object associated with the entry operation, which allows the partitioned region to store associated data together
- ▶ An exception terminates the operation and the exception is passed to the calling thread

### Enabling Custom Partitioning for Colocation

Use the `Key` or `callbackArgument` to implement the `PartitionResolver` interface, which enables custom partitioning. You can get the `Key` and the `callbackArgument` using the `EntryOperation` interface. Any of these can implement the `PartitionResolver` interface and provide a routing object (routing hint).

Configure your own `PartitionResolver` class in partition attributes (for example, if `Key` is a primitive type or string). If you want to colocate all trades by month and year, the key is implemented by the `TradeKey` class, which also implements the `PartitionResolver` interface.

The following example shows how to colocate all trade entries with the same month and year in the same region.

---

**Example 1.4 Sample Script to Colocate Data Entries on a Single Partitioned Region**


---

```

Public class TradeKey implements PartitionResolver {
private String tradeID;
private Month month;
private Year year;

public TradingKey(){
}
public TradingKey(Month month, Year year){
    this.month = month;
    this.year = year;
}
public Serializable getRoutingObject(EntryOperation opDetails){
    return this.month + this.year;
}
}

```

---

**Colocating Related Entries Across Multiple Partitioned Regions**

To colocate entries across multiple data regions, the application must perform the following:

- ▶ Configure the partitioned region to be colocated with another partitioned region, as shown in the following two examples. The first example is a declarative XML configuration, and the second example configures the partitioned region programmatically.
- ▶ Entries across data regions must return the same routing object (by implementing the `PartitionResolver` interface) for entries that have to be colocated.
- ▶ All entries that return the same routing object are colocated. For example, in a `CustomerOrders` partitioned region, all order entries that return the same `CustomerID` reside on the same member.

**Example 1.5 Colocating Data Entries on Multiple Partitioned Regions Declaratively**


---

```

<cache>
  <region name="Trades">
    <region-attributes>
      <partition-attributes>
        <partition-resolver="TradesPartitionResolver"> // Name
          <class-name>com.gemstone.gemfire.cache.TradesPartitionResolver
            </class-name>
        </partition-resolver>
      </partition-attributes>
    </region-attributes>
  </region>
  <region name="colocated_trade_history">
    <region-attributes>
      <partition-attributes colocated-with="Trades"> // COLOCATION ATTRIBUTE
        <partition-resolver="TradesPartitionResolver"> // Name
          <class-name>com.gemstone.gemfire.cache.TradesPartitionResolver
            </class-name>
        </partition-resolver>
      </partition-attributes>
    </region-attributes>
  </region>
</cache>

```

---

---

**Example 1.6 Colocating Data Entries on Multiple Partitioned Regions Programmatically**


---

```

// Create a new PartitionResolver based on ranges,
// so that one range can yield to bucket 1 and so on
PartitionResolver resolver = new TradesPartitionResolver();
// PartitionResolver extends Declarable so need to implement
init(Properties)
// method of Declarable;
//Set Partition resolver to partition attributes
PartitionAttributes attrs = new PartitionAttributesFactory().
setPartitionResolver(resolver).create();
//create a Trades Partition Region
Region Trades = new RegionFactory().setPartitionAttributes(attrs)
.create("Trades");
// Entry ops allowed before creation of associated partitioned regions.
Trades.put("foo", "bar");
// this Trades PR is now ready for operations
attrs = new PartitionAttributesFactory().setPartitionResolver(resolver)
.setColocatedWith(Trades.getFullPath()).create();
Region colocated_trade_history = new RegionFactory()
.setPartitionAttributes(attrs).create("colocated_trade_history");
colocated_trade_history.put("foo", "bar");
// even colocated_trade_history PR is now ready for operations

```

---

The following must occur while colocating related entries across multiple partitioned regions:

- ▶ To avoid an `IllegalStateException`, ensure that the region name passed in the `setColocatedWith` method has already been created.
- ▶ Colocated entities must enable custom partitioning to avoid an `IllegalStateException`.
- ▶ Colocated partitioned regions must have the same `PartitionResolver` object and return the same routing object.
- ▶ Colocated partitioned regions should have the same partition attributes, such as `numBuckets` and `redundantCopies`.

## Executing Functions on a Colocated Partitioned Region

**Example 1.7 Function Execution on a Colocated Partitioned Region**


---

```

class TradeCalc1 implements Function{
    public Serializable execute(FunctionContext context){
        Set keys = context.getFilter ();
// sub set of routing objects passed by the invoking client...
// only owned routing objects are passed
        <application logic ../>
        return <result>;
    }
    public String getId() {
        return this.getClass().getName();
    }
}

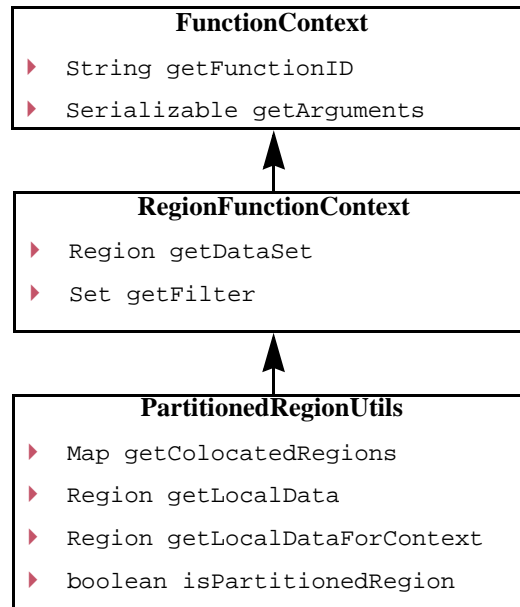
```

---

In the previous example, the execute method is passed in the FunctionContext. The FunctionContext is information made available to the application function that allows the function to get access to the data it needs to do its work on the node.

The following diagram shows the inheritance hierarchy and methods available through the FunctionContext.

**Figure 1.2 Hierarchy of Context Available to Function**



The following example provides an implementation to execute the function on a colocated partitioned region. The example colocates the entries made for the months of May and June in the same region.

**Example 1.8 Sample Script to Execute Function on a Colocated Partitioned Region**

```

PartitionedRegion r = null;
Serializable args = null;
Serializable result = null;

// Set the routing keys
// Function is a calculation on June and May trades.
Set<String> keys = new HashSet<String>
(Arrays.asList(new TradeKey(<june>, <2008>),
new TradeKey(<May>, <2008>)); // pseudo code

//Now execute ...
result = FunctionService.onRegion(r).withFilter(keys)
.execute("com.bigFatCompany.tradeService.cache.func.TradeCalc1.class")
.getResult();

// You can also pass arbitrary application arguments to the function ...
result = FunctionService.onRegion(r).withFilter(keys).withArgs(args)
.execute("com.bigFatCompany.tradeService.cache.func.TradeCalc1.class")
.getResult();
  
```

---

## Executing a Function Using a Custom Result Collector

The application can execute the same function and use a custom result collector instead of a generic result collector to process the results as desired. The following example provides the implementation of a custom result collector, `ArrayListResultCollector`.

### Example 1.9 Using a Custom Result Collector

---

```
//Providing a custom result collector
class ArrayListResultCollector implements ResultCollector {
    ArrayList list = new ArrayList();
    volatile boolean isClosed;
    //Add and gather results from each partition
    public synchronized void addResult(Serializable o) {
        this.list.addResult(o);
    }
    // Retrieve the final result
    public ArrayList getResult() throws FunctionException {
        return this.list;
    }
    // You can end the result collector, which will further cancel the
    // function execution running on partitions.
    // This can be useful when the function execution got timed out, or one
    // of the partition ran into hot loop or not able to respond back
    public void endResults() {
        isClosed = true;
    }
}
```

---

The application can use a custom result collector interface to sort or aggregate the result. You can perform aggregate functions like sum, minimum, maximum, and average on the results. The following example demonstrates the implementation of a custom result collector.

### Example 1.10 Execution of the Custom Result Collector

---

```
ResultCollector rc = new ArrayListResultCollector();
FunctionService.onRegion(r).withFilter(keys).withArgs(args).withCollector(rc)
)
.execute("com.bigFatCompany.tradeService.cache.func.TradeCalc1.class");
List resultList = (ArrayList)rc.getResult();
// Do something with the result
```

---

GemFire uses a thread pool to execute the function on partitions which are mapped using the filter provided on the dataset. If you do not specify any filters (routing keys) to the function execution service, the function is executed on all members.

## Data-Independent Functions

Data-independent functions are executed:

- ▶ On behalf of a client on either a single cache server member, or in parallel on all members of a *server group*.
- ▶ On behalf of a peer on either a single peer, or in parallel on all peers in the distributed system.

Executing a function on a single member is similar to executing stored procedures on database servers by applications. Data-independent functions are effective in the following instances:

- ▶ The application executes a server-side transaction or carries out data updates using the GemFire distributed lock service.
- ▶ The application initializes some components one time on each server. These components might be used later by executed functions.
- ▶ The application initializes and starts a third-party service, such as a messaging service.
- ▶ Any arbitrary aggregation operation that requires iteration over local data sets 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.

For more information about executing functions on one or more servers in a server group, see *Configuration With Server Grouping* in the *GemFire Enterprise Developer's Guide*.

### Scenarios For Executing Data-Independent Functions

You can execute data-independent functions in the following function execution service scenarios:

- ▶ On a single server
- ▶ On a server group
- ▶ On a single peer in a distributed system
- ▶ On all peers in a distributed system
- ▶ On a specified set of peers in a distributed system

The following sections provide details and `FunctionService` application program interfaces (APIs) to execute functions in the previously listed scenarios.

---

## Declaring and Registering the Function

Applications can declare and register the functions using declarative means (`cache.xml`) or through the API, as demonstrated in the examples below. All registered functions must have an *identifier*. Identifying functions allows the administrator to monitor the function activity and cancel the functions if required.

### Example 1.11 Registering Functions Declaratively Using XML

---

```
<cache>
  ...
  <function-service>
    <function>"com.bigFatCompany.tradeService.cache.func.TradeCalc1"
  </function>
    <function>"com.bigFatCompany.tradeService.cache.func.TradeCalc2"
  </function>
  </function-service>
  ...
</cache>
```

---

### Example 1.12 Registering a Function Programmatically Using an API

---

```
//Create Functions and register to FunctionService
Function function1 = new TradeCalc1(); //TradeCalc1 implements Function
interface
Function function2 = new TradeCalc2();
FunctionService.registerFunction(function1);
FunctionService.registerFunction(function2);
```

---

The following statements are true for all functions:

- ▶ Functions must be registered with each member before being executed.
- ▶ The ID that is returned from `Function.getId` can be any arbitrary string.
- ▶ Modifying a function instance after registration has no effect on function execution. If you want to execute the new function, you must register the new function with a different identifier.

## Executing Data-Independent Functions

### Executing the Function on a Single Server

The following example demonstrates the function execution service getting the free memory on a server in a pool.

```
API: Public static Execution onServer (Pool p)
```

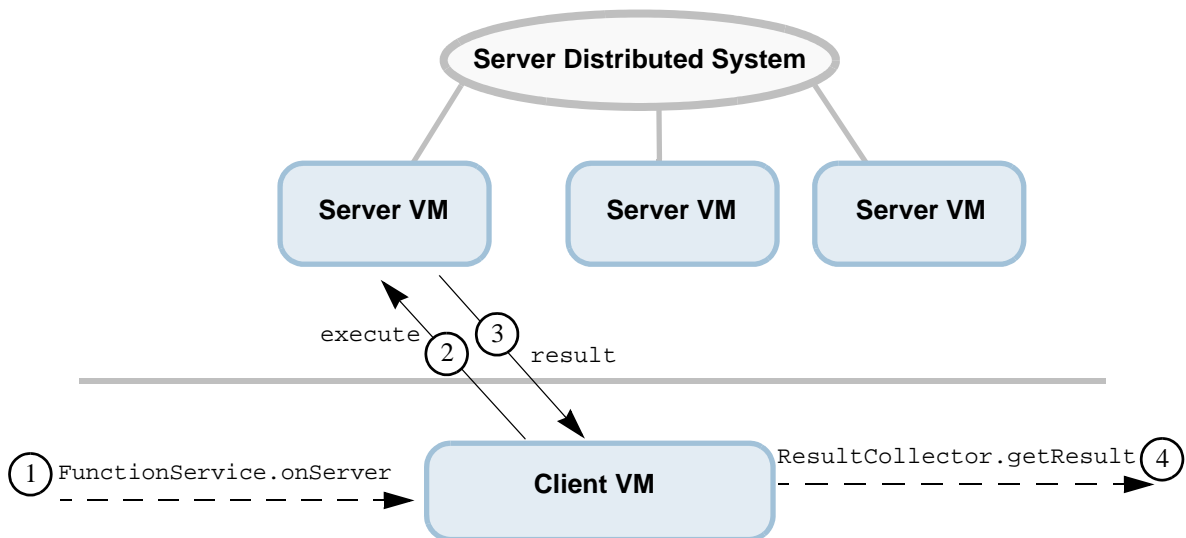
**Example 1.13 Data-Aware Function from a Client on a Single Server**

```

Pool pool ;
Serializable args ;
Function getFreeMemory ;
ResultCollector rc = FunctionService.onServer(pool)
.withCollector(new MyCustomResultCollector())
.withArgs(args)
.execute(getFreeMemory);
// Application can do something else here
Serializable functionResult = rc.getResult();

```

The following figure graphically illustrates the sequence of events that occur when a data-aware function is executed from a client on a single server:

**Figure 1.3 Data-Aware Function from a Client on a Single Server****Executing the Function on a Server Group**

The following example demonstrates the function execution service getting the free memory on all servers in a pool.

```
API: Public static Execution onServers (Pool p)
```

**Example 1.14 Data-Aware Function from a Client on a Server Group**

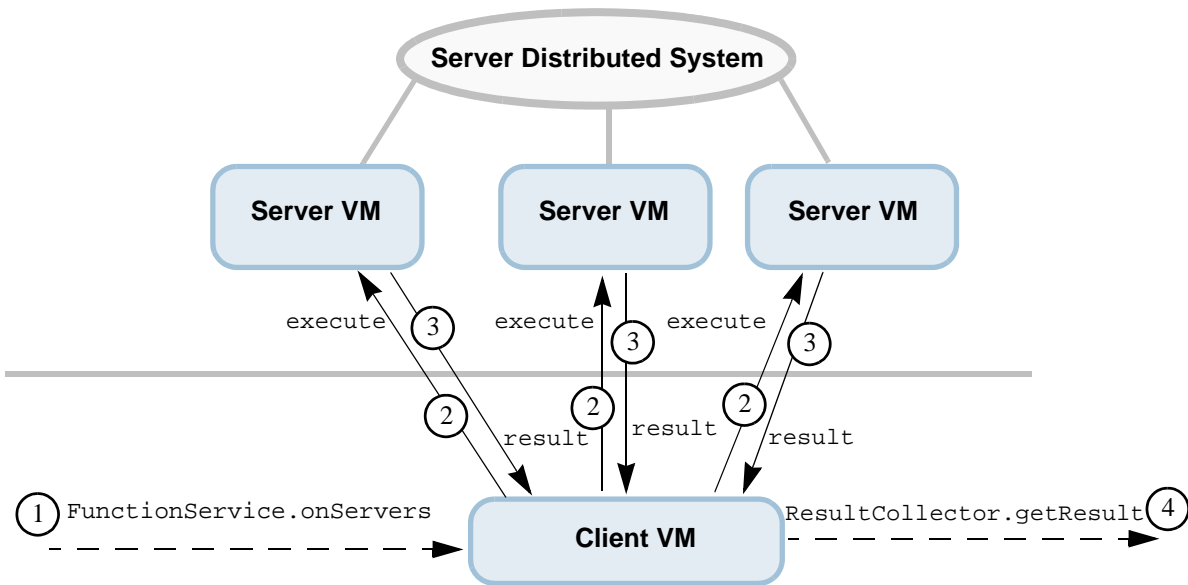
```

Pool pool ;
Serializable args ;
Function getFreeMemory ;
ResultCollector rc = FunctionService.onServers(pool)
.withCollector(new MyCustomResultCollector())
.withArgs(args)
.execute(getFreeMemory);
// Application can do something else here
Serializable functionResult = rc.getResult();

```

The following figure graphically illustrates the sequence of events that occur when a data-aware function is executed from a client on a server group:

**Figure 1.4 Data-Aware Function from a Client on a Server Group**



### Executing the Function on Single Member in a Distributed System

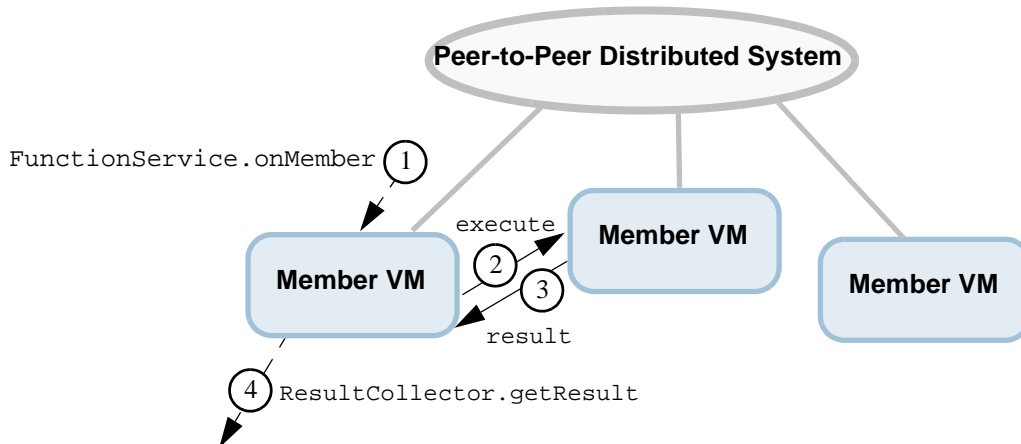
The following example demonstrates the function execution service getting the free memory on a single member in a distributed system.

```
API: Public static Execution onMember(DistributedSystem s, DistributedMember m)
```

### Example 1.15 Data-Aware Function from a Client on Single Peer in a System

```
Serializable args ;
Function getFreeMemory ;
DistributedSystem ds ;
DistributedMember member ;
ResultCollector rc = FunctionService.onMember(ds, member)
.withCollector(MyCustomResultCollector())
.withArgs(args)
.execute(getFreeMemory);
// Application can do something else here
Serializable functionResult = rc.getResult();
```

The following figure graphically illustrates the sequence of events that occur when a data-aware function is executed from a client on a single member:

**Figure 1.5 Data-Aware Function from a Client on a Single Member****Executing the Function on all Members in a Distributed System**

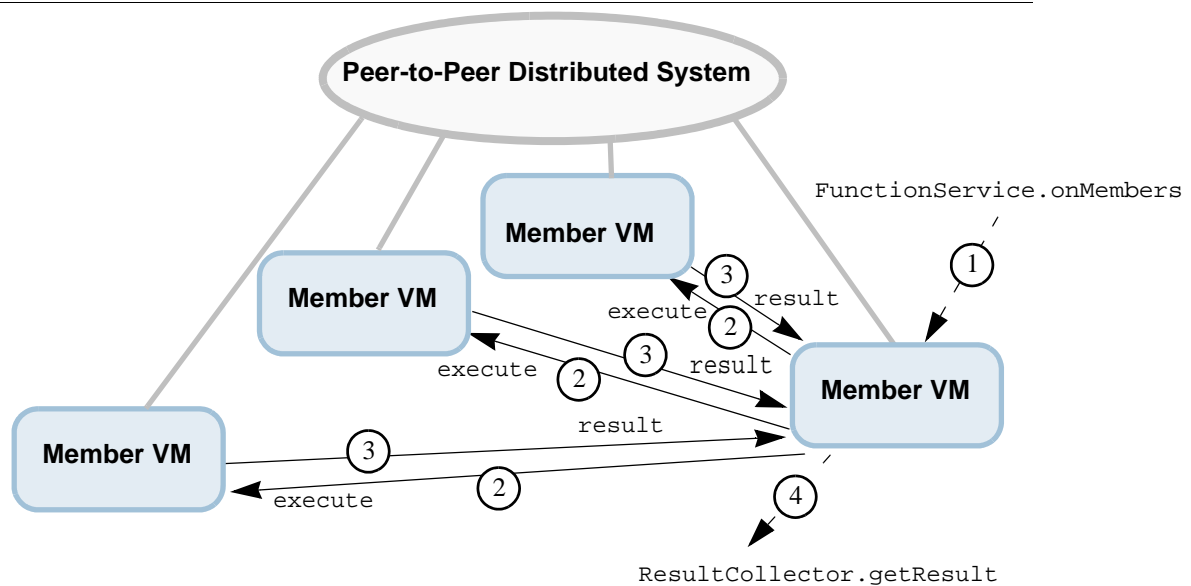
The following example demonstrates the function execution service getting the free memory on all members in a distributed system.

API: `Public static Execution onMembers (DistributedSystem s)`

**Example 1.16 Data-Aware Function from a Client on all Peers in a System**

```

Serializable args ;
Function getFreeMemory ;
DistributedSystem ds ;
ResultCollector rc = FunctionService.onMembers(ds)
.withCollector(MyCustomResultCollector())
.withArgs(args)
.execute(getFreeMemory);
// Application can do something else here
Serializable functionResult = rc.getResult();
  
```

**Figure 1.6 Data-Aware Function from a Client on all Members****Executing the Function on a Specified set of Members in a Distributed System**

The following example demonstrates the function execution service getting the free memory on a specific set of members in a distributed system.

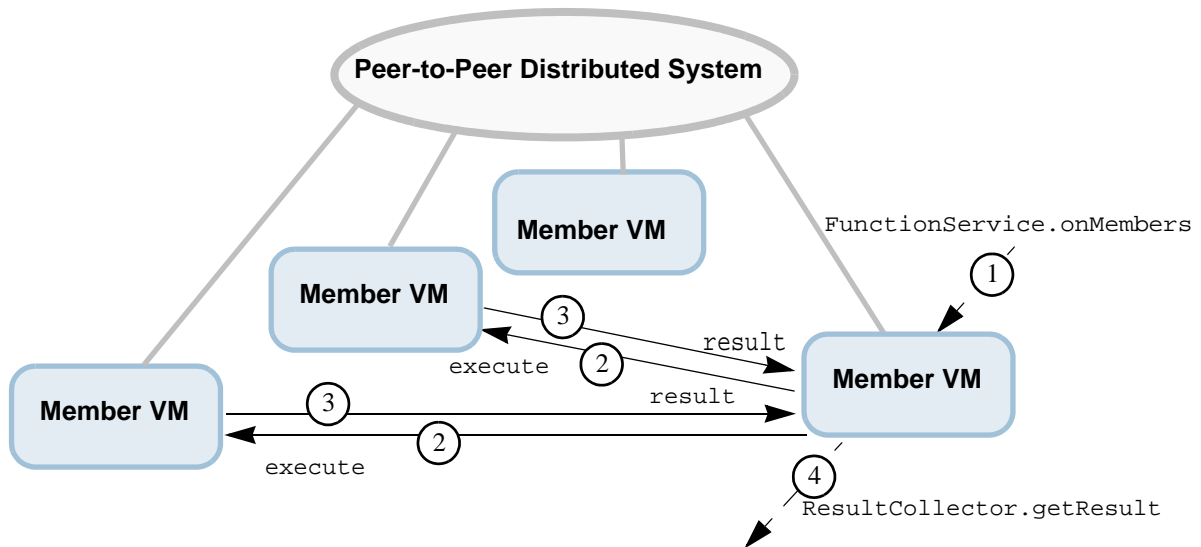
API: `Public static Execution onMembers (DistributedSystem s, Set members)`

**Example 1.17 Data-Aware Function from a Client on Specific Peers in a System**

```

Serializable args ;
Function getFreeMemory ;
DistributedSystem ds ;
Set memberSet = new HashSet();
DistributedMember member1 ;
DistributedMember member2 ;
memberSet.add(member1);
memberSet.add(member2);
ResultCollector rc = FunctionService.onMembers(ds,memberSet)
.withCollector(MyCustomResultCollector())
.withArgs(args)
.execute(getFreeMemory);
// Application can do something else here
Serializable functionResult = rc.getResult();

```

**Figure 1.7 Data-Aware Function from a Client on Specified Members**

## Failover During Function Execution

When a function is invoked from a client as either a data-independent or data-dependent function, and the original request fails due to a server-side issue (the cache server goes down, or it loses its connection), the GemFire client automatically retries the function execution. Since the function execution service is not transactional, this can result in the same function executed again that was partially executed earlier in the initial failed attempt.

## Solutions and Use Cases

The function execution service provides solutions for the following application use cases:

- ▶ An application intending to execute a server-side transaction or carry out data updates using the GemFire distributed locking service.
- ▶ An application wanting to initialize 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.

---

## Eviction and Overflow in Partitioned Regions - Early Access

Partitioned regions (PR) now support the eviction and overflow of least-recently-used (LRU) entries based on user-specified capacity limits. Eviction and overflow is desirable when the aggregate amount of space taken up by the partitioned region across the grid exceeds the available memory and each grid member requires strategies to manage its memory usage. Eviction removes an entry, and overflow moves least-recently-used entries to disk so they can still be retrieved.

*Eviction and overflow for partitioned regions is not documented in the manuals accompanying this release. This new functionality augments existing information in the GemFire Enterprise Developer's Guide and the GemFire Enterprise System Administrator's Guide, and nullifies any statements about partitioned regions not supporting eviction and overflow.*

In most cases, eviction and overflow for partitioned regions uses the same interface and semantics for eviction and overflow used by other types of regions. The following descriptions of the configuration settings and options are provided for reference. See the *GemFire Enterprise Developer's Guide* for more information about configuring eviction and overflow.

There are two different actions (based on the DTD) that occur when eviction is enabled. The following use-cases are supported with PR:

- ▶ **Overflow-to-disk action:** It is acceptable to retain the entire key space in memory. No entries are to be lost (destroyed). It is acceptable that some values may be expensive to retrieve since they are on disk. For more information about the overflow mechanism, see *Overflowing to Disk* in the *GemFire Enterprise® Developer's Guide*.

For example, the region keys represent sales customers. Values for each customer include information such as contact information and sales history. Only the most active (recent) customers' information must be readily accessible, but all customers must be accessible.

- ▶ **Local-destroy action:** The amount of space taken up by the PR keys plus values exceeds available memory. It is acceptable for entries to be lost (locally destroyed) because the region is strictly a cache; that is, there is a cache loader or other mechanism so that the user's schema is not corrupted by the loss of data.

For example, the region keys represent sales customers. Values for each customer include information such as contact information and sales history. These values are maintained by a database of records, and the region has a `CacheWriter` and a `CacheLoader` to manage updates and loads. The data shows a good working set behavior so that most reads can be satisfied without using the underlying database.

## Setting Eviction Attributes for a Partitioned Region

The `EvictionAttributes` are created and added to the `RegionAttributes`. While creating eviction attributes, consider the following important factors:

- ▶ The eviction algorithm sets the conditions and limits where entry eviction occurs. These are the three eviction algorithms you can choose from, shown as XML properties:
  - ▶ `lru-heap-percentage`—Considers the VM heap size before performing an eviction on the leastrecently-used entries. The `lru-heap-percentage` takes into account the percentage used by each JVM's heap. The heap size limit can be different for each VM.
  - ▶ `lru-entry-count`—Considers the number of entries in the VM before performing an eviction. This value must be the same for all members of a PR, accessors and datastore alike.
  - ▶ `lru-memory-size`—Considers the amount of memory bytes consumed by the region before performing an eviction. The memory size maximum value can be different for each VM.

*The maximum value should not exceed the PR `localMaxMemory` configuration. If it does, it is automatically reset to the `localMaxMemory` value and a warning is logged indicating the reset condition.*

- ▶ Eviction actions determine what to do with the entries that are being evicted. These are the options, shown as XML properties:
  - ▶ `local-destroy`—The LRU region entry is locally destroyed.
  - ▶ `overflow-to-disk`—The LRU region entry value is written to disk.

*To ensure reliable read behavior across the partitioned region, it is recommended that you use `overflow-to-disk` instead of `local-destroy` for eviction.*

### Setting Eviction Attributes With `Local_Destroy` Action

The following declarative and programmatic examples demonstrate setting up entry LRU-based eviction attributes.

#### Example 1.18 Declarative PR Entry LRU-based Eviction Using the `Local_Destroy` Option

```
<region name="demoPR">
  <region-attributes>
    <partition-attributes local-max-memory="50" redundant-copies="1"
      total-num-buckets="13" />
    <eviction-attributes>
      <lru-entry-count maximum="100" action="local-destroy"/>
    </eviction-attributes>
  </region-attributes>
</region>
```

**Example 1.19 Programmatic PR Entry LRU-based Eviction Using the Local\_Destroy Option**


---

```

PartitionAttributesFactory paf = new PartitionAttributesFactory();
PartitionAttributes prAttr = paf.setRedundantCopies(0).setLocalMaxMemory(50)
.setTotalNumBuckets(13).create();
EvictionAttributes evictionAttributes =
EvictionAttributes.createLRUEntryAttributes(maxEntries,
EvictionAction.LOCAL_DESTROY);
AttributesFactory attr = new AttributesFactory();
    attr.setPartitionAttributes(prAttr);
    attr.setEvictionAttributes(evictionAttributes);
this.region = this.cache.createRegion("demoPR", attr.create());

```

---

The following declarative and programmatic examples demonstrate setting up heap LRU eviction with the heap percentage set to a maximum of 66%, with local-destroy as the eviction action.

**Example 1.20 Declarative PR Heap LRU Based Eviction Using the Local\_Destroy Option**


---

```

<region name="demoPR">
  <region-attributes>
    <partition-attributes local-max-memory="50" redundant-copies="1"
      total-num-buckets="13" />
    <eviction-attributes>
      <lru-heap-percentage action="local-destroy" maximum="66"
        time-interval="100"/>
    </eviction-attributes>
  </region-attributes>
</region>

```

---

**Example 1.21 Programmatic PR Heap LRU Based Eviction Using the Local\_Destroy Option**


---

```

//Creating Partition Attributes
PartitionAttributesFactory paf = new PartitionAttributesFactory();
PartitionAttributes prAttr = paf.setRedundantCopies(1)
.setLocalMaxMemory(50).setTotalNumBuckets(13).create();
//Creating Eviction Attributes
EvictionAttributes evictionAttributes = EvictionAttributes
.createLRUHeapAttributes(66 /* heapPercentage */,
100/* evictorInterval */, EvictionAction.LOCAL_DESTROY);
AttributesFactory attr = new AttributesFactory();
attr.setPartitionAttributes(prAttr);
attr.setEvictionAttributes(evictionAttributes);
Region region = cache.createRegion("demoPR", attr.create());

```

---

While you are setting up memory LRU-based eviction attributes, ensure that:

- ▶ the evictionSizeInMB is greater than the total number of buckets in a partitioned region.
- ▶ the maximum value of evictionSizeinMB can be localMaxMemory of the partitioned region.

The following declarative and programmatic examples demonstrate setting up memory LRU-based eviction attributes.

**Example 1.22 Declarative PR Memory LRU-based Eviction Using the Local\_Destroy Option**

```

<region name="demoPR">
  <region-attributes>
    <partition-attributes local-max-memory="512" redundant-copies="1"
      total-num-buckets="13" />
    <eviction-attributes>
      <lru-memory-size maximum="256" action="local-destroy" >
        <class-name>com.gemstone.gemfire.cache.util.ObjectSizerImpl
        </class-name>
      </eviction-attributes>
    </region-attributes>
  </region>

```

**Example 1.23 Programmatic PR Memory LRU-based Eviction Using the Local\_Destroy Option**

```

//Creating Partition Attributes
PartitionAttributesFactory paf = new PartitionAttributesFactory();
PartitionAttributes prAttr = paf.setRedundantCopies(1)
    .setLocalMaxMemory(512).setTotalNumBuckets(13).create();
//Creating Eviction Attributes
EvictionAttributes evictionAttributes = EvictionAttributes
    .createLRUMemoryAttributes(256 /* maximumMegabytes */
        , new ObjectSizerImpl(), EvictionAction.LOCAL_DESTROY);
AttributesFactory attr = new AttributesFactory();
attr.setPartitionAttributes(prAttr);
attr.setEvictionAttributes(evictionAttributes);
Region region = cache.createRegion("demoPR", attr.create());

```

**Setting Eviction Attributes With Overflow\_to\_Disk Action**

The following declarative and programmatic examples demonstrate setting entry LRU-based eviction attributes when the action is `Overflow_To_Disk`.

**Example 1.24 Declarative PR Entry LRU-based Eviction Using the Overflow\_to\_Disk Option**

```

<region name="demoPR">
  <region-attributes>
    <partition-attributes local-max-memory="50" redundant-copies="1"
      total-num-buckets="13" />
    <eviction-attributes>
      <lru-entry-count maximum="100" action="overflow-to-disk"/>
    </eviction-attributes>
    <disk-write-attributes>
      <synchronous-writes/>
    </disk-write-attributes>
    <disk-dirs>
      <disk-dir>overflowDir</disk-dir>
    </disk-dirs>
  </region-attributes>
</region>

```

**Example 1.25 Programmatic PR Entry LRU-based Eviction Using the Overflow\_to\_Disk Option**


---

```

// Creating Partition Attributes
PartitionAttributesFactory paf = new PartitionAttributesFactory();
PartitionAttributes prAttr = paf.setRedundantCopies(1)
    .setLocalMaxMemory(50).setTotalNumBuckets(13).create();
// Creating Eviction Attributes
EvictionAttributes evictionAttributes =
EvictionAttributes.createLRUEntryAttributes(
    100/*maxEntries*/, EvictionAction.OVERFLOW_TO_DISK);
AttributesFactory attr = new AttributesFactory();
//Creating Disk Attributes as eviction action is overflow-to-disk
DiskWriteAttributesFactory dwaf = new DiskWriteAttributesFactory();
dwaf.setSynchronous(true);
attr.setDiskWriteAttributes(dwaf.create());
File[] diskDirs = new File[1];
diskDirs[0] = new File("overflowDir");
diskDirs[0].mkdirs();
attr.setDiskDirs(diskDirs);
attr.setPartitionAttributes(prAttr);
attr.setEvictionAttributes(evictionAttributes);
Region region = this.cache.createRegion("demoPR", attr.create());

```

---

The following declarative and programmatic examples demonstrate setting up heap LRU eviction with the heap percentage set to a maximum of 66%, with `overflow-to-disk` as the eviction action.

**Example 1.26 Declarative PR Heap LRU-based Eviction Using the Overflow\_to\_Disk Option**


---

```

<region name="demoPR">
  <region-attributes>
    <partition-attributes local-max-memory="50" redundant-copies="1"
      total-num-buckets="13" />
    <eviction-attributes>
      <lru-heap-percentage action="overflow-to-disk" maximum="66"
        time-interval="100"/>
    </eviction-attributes>
    <disk-write-attributes>
      <synchronous-writes/>
    </disk-write-attributes>
    <disk-dirs>
      <disk-dir>overflowDir</disk-dir>
    </disk-dirs>
  </region-attributes>
</region>

```

---

**Example 1.27 Programmatic PR Heap LRU-based Eviction Using the Overflow\_to\_Disk Option**

```

final int heapPercentage = 40; // Threshold for heapSize, Entries shall be
evicted once heap usage reaches heapPercentage
final int evictorInterval = 500; // Time period to check heap size, in this
case it would check every 100 ms
PartitionAttributesFactory paf = new PartitionAttributesFactory();
PartitionAttributes prAttr = paf.setRedundantCopies(1).setLocalMaxMemory(50)
.setTotalNumBuckets(13).create();
EvictionAttributes evictionAttributes =EvictionAttributes
.createLRUHeapAttributes(heapPercentage, evictorInterval,
EvictionAction.OVERFLOW_TO_DISK);
AttributesFactory attr = new AttributesFactory();
attr.setPartitionAttributes(prAttr);
attr.setEvictionAttributes(evictionAttributes);
this.region = this.cache.createRegion("demoPR", attr.create());

```

While you are setting up the memory LRU-based eviction attributes, ensure that:

- ▶ the evictionSizeInMB is greater than the total number of buckets in a partitioned region.
- ▶ the maximum value of evictionSizeinMB can be localMaxMemory of the partitioned region.

The following declarative and programmatic examples demonstrate setting up memory LRU-based eviction attributes.

**Example 1.28 Declarative PR Memory LRU-based Eviction Using the Overflow\_to\_Disk Option**

```

<region name="demoPR">
  <region-attributes>
    <partition-attributes local-max-memory="512" redundant-copies="1"
      total-num-buckets="13"/>
    <eviction-attributes>
      <lru-memory-size maximum="256" action="overflow-to-disk" >
        <class-name>com.gemstone.gemfire.cache.util.ObjectSizerImpl
        </class-name>
      </lru-memory-size>
    </eviction-attributes>
    <disk-write-attributes>
      <synchronous-writes/>
    </disk-write-attributes>
    <disk-dirs>
      <disk-dir>overflowDir</disk-dir>
    </disk-dirs>
  </region-attributes>
</region>

```

---

**Example 1.29 Programmatic PR Memory LRU-based Eviction Using the Overflow\_to\_Disk Option**

---

```
//Creating Partition Attributes
PartitionAttributesFactory paf = new PartitionAttributesFactory();
PartitionAttributes prAttr = paf.setRedundantCopies(1)
    .setLocalMaxMemory(512).setTotalNumBuckets(13).create();
//Creating Eviction Attributes
EvictionAttributes evictionAttributes = EvictionAttributes
    .createLRUMemoryAttributes(256 /* maximumMegabytes */
        , new ObjectSizerImpl(), EvictionAction.OVERFLOW_TO_DISK);
AttributesFactory attr = new AttributesFactory();
//Creating Disk Attributes as eviction action is overflow-to-disk
DiskWriteAttributesFactory dwaf = new DiskWriteAttributesFactory();
dwaf.setSynchronous(true);
attr.setDiskWriteAttributes(dwaf.create());
File[] diskDirs = new File[1];
diskDirs[0] = new File("overflowDir");
diskDirs[0].mkdirs();
attr.setDiskDirs(diskDirs);
attr.setPartitionAttributes(prAttr);
attr.setEvictionAttributes(evictionAttributes);
Region region = cache.createRegion("demoPR", attr.create());
```

---

---

## Expiration in Partitioned Regions - Early Access

Expiration can now be performed on partitioned regions (PR). Expiration ensures that stale data is not fetched from the cache.

*Expiration for partitioned regions is not documented in the manuals accompanying this release. This new functionality augments existing information in the GemFire Enterprise Developer's Guide and the GemFire Enterprise System Administrator's Guide, and nullifies any statements about partitioned regions not supporting expiration.*

In most cases, expiration for partitioned regions uses the same interface and semantics for expiration used by other types of regions. The following descriptions of the configuration settings and options are provided for reference. See the *GemFire Enterprise Developer's Guide* for more information about configuring expiration.

When you create expiry attributes, you determine the level of data that must be expired. Expiry occurs at two levels:

- ▶ **Entry level** expiry—the expiration action is performed on data entries
- ▶ **Region level** expiry—the expiration action is performed on the region

Then, determine if the expiration occurs on entry or region expiration by specifying the `time-to-live` or `idle-time` settings. These settings can be configured declaratively using a `cache.xml` file, or they can be set programmatically. This is a description of the entry-level and region-level expiration settings configurable in `cache.xml`:

- ▶ `entry-time-to-live`—Specifies how long the region's entries can remain unexpired in the cache without anyone accessing or updating them
- ▶ `entry-idle-time`—Specifies how long the region's entries can remain unexpired in the cache without anyone accessing them
- ▶ `region-time-to-live`—Specifies how long the region can remain unexpired in the cache without anyone accessing or updating it
- ▶ `region-idle-time`—Specifies how long the region can remain unexpired in the cache without anyone accessing it

*To ensure reliable read behavior across the partitioned region, it is recommended that you use `entry-time-to-live` or `region-time-to-live` for expiration.*

After determining which level and how the data must be expired, choose the eviction action that must be performed when an entry or region expires. One of the following actions can be performed:

- ▶ **invalidate**—Sets the value to null
- ▶ **destroy**—Removes the data

## Enabling Statistics for Expiration

You need to enable statistics to perform expiration for a partitioned region. This `cache.xml` snippet shows how to enable statistics:

```
<region name="data">
  <region-attributes statistics-enabled="true">
    <partition-attributes .../>
  </region-attributes>
</region>
```

Alternatively, you can enable statistics by setting these properties in the `gemfire.properties` file:

```
statistic-sampling-enabled=true
enable-time-statistics=true
```

The following declarative and programmatic examples demonstrate setting entry-level expiry when the expiry action is `destroy` using the `entry-time-to-live` property.

### Example 1.30 Declarative PR Entry Level Expiry Using the Destroy Option

---

```
<region name="demoPR">
  <region-attributes statistics-enabled="true">
    <partition-attributes local-max-memory="50" redundant-copies="1"
      total-num-buckets="13" />
    <entry-time-to-live>
      <expiration-attributes timeout="60" action="destroy"/>
    </entry-time-to-live>
  </region-attributes>
</region>
```

---

### Example 1.31 Programmatic PR Entry Level Expiry Using the Destroy Option

---

```
// Creating Partition Attributes
PartitionAttributesFactory paf = new PartitionAttributesFactory();
PartitionAttributes prAttr = paf.setRedundantCopies(1)
    .setLocalMaxMemory(50).setTotalNumBuckets(13).create();
// Creating Expiration Attributes
ExpirationAttributes expiration = new ExpirationAttributes(
    60 /* expirationTime in seconds */, ExpirationAction.DESTROY);
AttributesFactory attr = new AttributesFactory();
attr.setEntryTimeToLive(expiration);
attr.setPartitionAttributes(prAttr);

Region region = cache.createRegion("demoPR", attr.create());
```

---

The following examples demonstrate setting entry-level expiry when the expiry action is invalidate using the `entry-idle-time` property.

### Example 1.32 Declarative PR Entry Level Expiry Using the Invalidate Option

```
<region name="demoPR">
  <region-attributes statistics-enabled="true">
    <partition-attributes local-max-memory="50" redundant-copies="1"
      total-num-buckets="13" />
    <entry-idle-time>
      <expiration-attributes timeout="60" action="invalidate"/>
    </entry-idle-time>
  </region-attributes>
</region>
```

### Example 1.33 Programmatic PR Entry Level Expiry Using the Invalidate Option

```
// Creating Partition Attributes
PartitionAttributesFactory paf = new PartitionAttributesFactory();
PartitionAttributes prAttr = paf.setRedundantCopies(1)
    .setLocalMaxMemory(50).setTotalNumBuckets(13).create();
// Creating Expiration Attributes
ExpirationAttributes expiration = new ExpirationAttributes(
    60 /* expirationTime in seconds */, ExpirationAction.INVALIDATE);
AttributesFactory attr = new AttributesFactory();
attr.setEntryIdleTimeout(expiration);
attr.setPartitionAttributes(prAttr);
Region region = cache.createRegion("demoPR", attr.create());
```

## Content-Based Expiration for Partitioned Regions

In a partitioned region that supports expiration, the length of time (either `entry-time-to-live` or `entry-idle-time`) must not be the same for all entries. Also, you can define a way to customize the expiration on a per-entry basis. The customization must be a function of either the key or the value of the entry.

The following example shows the DTD for custom expiration attributes:

### Example 1.34 Defining Custom Expiration Attributes

```
<!ELEMENT expiration-attributes (custom-expiry?)>
<!ELEMENT custom-expiry (
  class-name,
  parameter)>
<!ATTLIST custom-expiry
```

*A `local_destroy` or `local_invalidate` action generates an `IllegalStateException` if used on a replicated region.*

---

The following example shows how to set custom expiry for partitioned regions.

**Example 1.35 Setting Custom Expiry for Partitioned Regions**

---

```
PartitionAttributesFactory paf = new PartitionAttributesFactory();
paf.setRedundantCopies(1).setTotalNumBuckets(12);
PartitionAttributes prAttr = paf.create();
RegionAttributes attrs = null;
AttributesFactory fac = new AttributesFactory();
ArrayList keyList = new ArrayList();
for(int i=0; i< 10; i++)
    keyList.add("key" + i);
CustomExpiry cusExp = new TestExpiry(keyList, expiration) ;
fac.setCustomEntryTimeToLive(cusExp);
fac.setCustomEntryIdleTimeout(cusExp);
fac.setStatisticsEnabled(true);
fac.setPartitionAttributes(prAttr);
attrs = fac.create();
Region pr = cache.createRegion(partitionedRegionName, attrs);
```

---

---

## Region Snapshots - Early Access

Snapshots are managed through the `saveSnapshot` and `loadSnapshot` methods of the `Region` interface. These are the key considerations for snapshots:

- ▶ Taking a snapshot does not stop region activity, so modifications to the region that are made while the snapshot is being taken may or may not be saved to disk.
- ▶ Snapshots are not recursive. The region's entries are saved, but subregions and subregion entries are not.
- ▶ Loading a snapshot removes all existing distributed region contents before loading the data from disk. All existing subregions and entries are destroyed before the snapshot is loaded. Remote regions with the same name are cleared of all subregions and entries, and remote regions that are configured as replicas do a new `getInitialImage` operation to get the data from this snapshot. All existing references to the distributed region become unusable and, if used, throw a `RegionReinitializedException`. All applications with references to the region must reacquire the region using `Cache.getRegion` or `Region.getSubregion`.

For additional information on snapshots, see the online Java documentation.

## EntryEvent Get Serialized Values - Early Access

You can retrieve serialized values from `EntryEvent`. This is useful if you get values from one region's events just to put them into a separate cache region. The standard `getOldValue` and `getNewValue` functions deserialize the values, which would then be reserialized when you put them into the cache. There is no counterpart `put` function as the `put` recognizes that the value is serialized and bypasses the serialization step.

For details on the new API supporting this functionality, see the online Java documentation in `com.gemstone.gemfire.cache` for the interface `SerializedCacheValue` and for the `EntryEvent` methods `getSerializedNewValue` and `getSerializedOldValue`.

## Bulk Operations Using `getAll` - Early Access

The `Region.getAll` method gets values for all keys in the input collection. The method returns a map of values for the input keys. See the `Region` interface description in the Java online API documentation for details about using `getAll`.

## Load Balancing in Multi-site Installations - Early Access

GemFire Enterprise now allows you to balance the load of gateway communication between multiple VMs. The basic approach is to define multiple gateway hubs and then split outgoing region event messaging between them.

With the new functionality, you can:

- ▶ Specify multiple gateway-hubs in your cache
- ▶ Indicate for each VM whether a hub should start as primary or secondary
- ▶ Point each gateway-enabled region to a specific gateway-hub

Only the primary hub instances send region events to remote sites. By splitting your hub primaries between multiple VMs and splitting your region events between hubs, you can achieve a level of load balancing in your outgoing hub messaging.

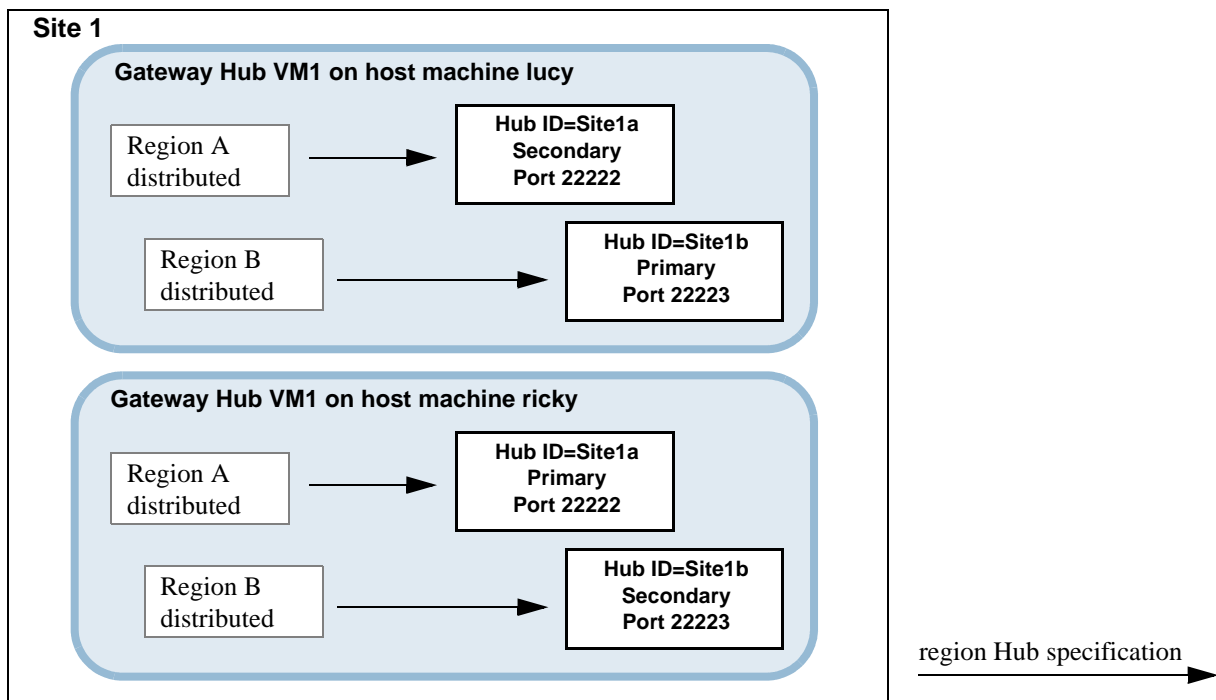
### The Multiple-Hub Configuration

Figure 1.1 shows a typical multiple-hub configuration. There may be any number of VMs in this distributed system in addition to the ones that host the gateway hubs.

In this configuration:

- ▶ There are two VMs hosting gateway hubs.
- ▶ There are two gateway hubs in each hub VM. One hub is primary in VM1 and the other hub is primary in VM2. Only the primaries send data to remote sites.
- ▶ Each hub has some of the region traffic pointing to it.

**Figure 1.8 Two-Hub Configuration With Primaries in Different VMs**

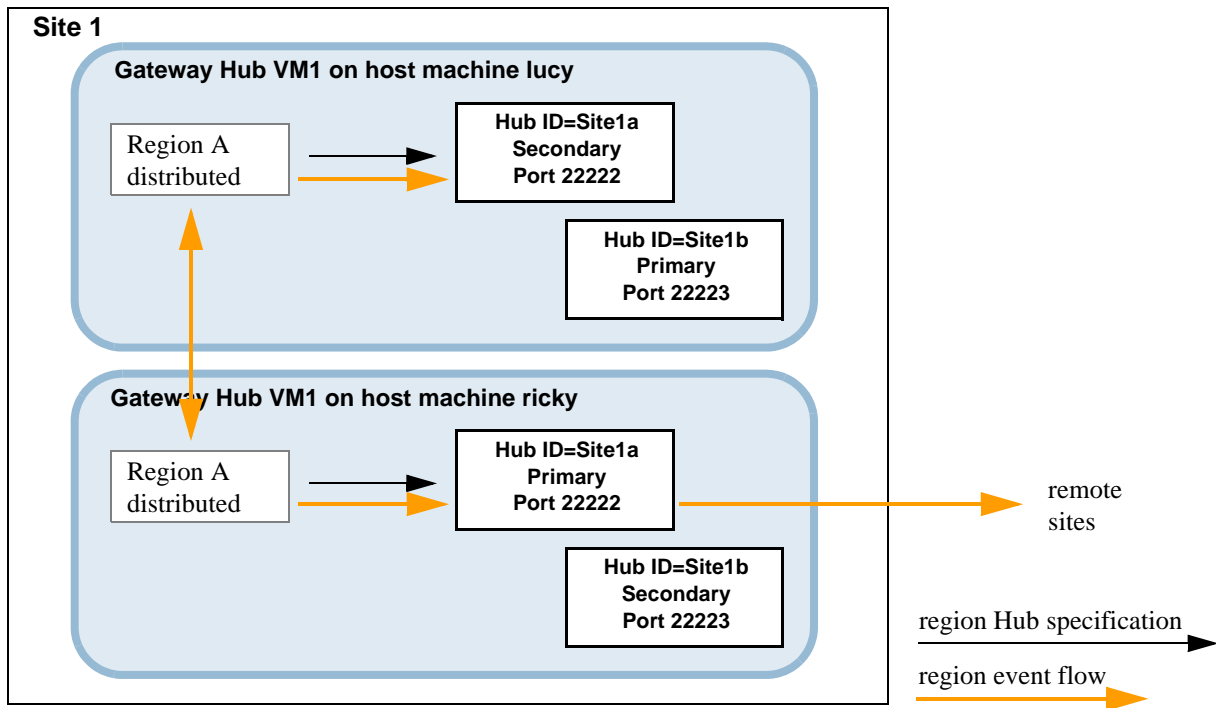


## Event Flow With Multiple Hubs

Figure 1.2 shows event flow for region A in the multiple-hub configuration of [Figure 1.8 on page 31](#).

- ▶ Region events originating in VM1 are stored in the local secondary Hub Site1a and are also distributed to VM2. Then in VM2, the events arriving in the local Region are sent to the VM's primary Hub Site1a, which then forwards them to remote sites.
- ▶ Region events originating in VM2 are sent directly out of the local primary Hub Site1a. They are also distributed to VM1 for cache update and storage in the secondary Hub Site1a.
- ▶ HubSite1b never sees this region's outgoing events.

**Figure 1.9 Outgoing Region Event Flow for Two-Hub Configuration**



*If you have multiple hubs, make sure you explicitly assign one hub to each distributed region using the `hub-id` region attribute. If you do not specify a hub for a region, its events will go to all hubs and each hub's primary will send the message to the remote sites, resulting in duplicate sends.*

## Configuring the Multiple-Hub Site

These tables list the new configuration attributes for multi-site load balancing.

**Table 1.1 Gateway Hub Attribute**

gateway-hub attribute	Description	Default
startup-policy	Specifies whether the hub should attempt to start as primary or secondary hub. Valid values are <code>primary</code> , <code>secondary</code> , and <code>none</code> .	none

**Table 1.2 Region Attribute**

region attribute	Description	Default
hub-id	Specifies the destination hub for region events. <i>For multiple-hub systems this must be specified. If it is not, region events go to all available hubs, which results in duplicate sends to remote sites.</i>	" "

### Hub Configuration

The `cache.xml` in this example defines the gateway hub configurations for VM1 in [Figure 1.8 on page 31](#).

**Example 1.36 Two-Hub Gateway Hub Configuration in `cache.xml`**

```
<cache>
  <gateway-hub id="Site1a" port="22222" startup-policy="secondary">
    ...
  </gateway-hub>
  <gateway-hub id="Site1b" port="22223" startup-policy="primary">
    ...
  </gateway-hub>
```

This shows the API equivalent.

**Example 1.37 Two-Hub Gateway Hub Configuration Through the API**

```
// Create or obtain the GemFire cache
Cache cache = ... ;

// Create the first Gateway Hub
GatewayHub site1aHub = cache.setGatewayHub("Site1a", 22222);
site1aHub.setStartupPolicy("STARTUP_POLICY_SECONDARY");

// Create the second Gateway Hub
GatewayHub site1bHub = cache.setGatewayHub("Site1b", 22223);
site1bHub.setStartupPolicy("STARTUP_POLICY_PRIMARY");

// Define gateways and endpoints for the gateway hubs ...
```

## Region Configuration

When you use multiple hubs, you must point each gateway-enabled region at one of the hubs. If you do not specify a hub, region events are delivered to every hub and then sent to remote sites by each hub's primary instance. The region configuration must be consistent across the distributed system, even in caches where there are no hubs running.

Split region events between your hubs as evenly as you can. If you have two hubs and three regions, for example, with one region having the bulk of the events, put that region on one hub and the others on the other hub.

This `cache.xml` assigns the example regions in [Figure 1.8 on page 31](#) to their respective hubs.

### Example 1.38 Enabling Region Communication With Gateway Hubs in `cache.xml`

```
<region name="A">
  <region-attributes ... enable-gateway="true" hub-id="Sitela"/>
  ...
</region>
<region name="B">
  <region-attributes ... enable-gateway="true" hub-id="Sitelb"/>
  ...
</region>
```

This is the same configuration through the API.

### Example 1.39 Enabling Region Communication With a Gateway Hub Through the API

```
// Region A
AttributesFactory factory = new AttributesFactory();
factory.setEnableGateway(new Boolean(true));
factory.setGatewayHubId("Sitela");
Region multiSiteRegionA = cache.createRegion("A", factory.create());

// Region B
AttributesFactory factory = new AttributesFactory();
factory.setEnableGateway(new Boolean(true));
factory.setGatewayHubId("Sitelb");
Region multiSiteRegionB = cache.createRegion("B", factory.create());
```

## How Hub Startup Policy Affects Startup Behavior

Every hub must have one primary instance running to maintain communication with remote sites. The other instances are secondaries that act as backups to the primary.

You can specify which hub instances should start as primary and which as secondary in your distributed system. When a hub is initialized, it attempts to assume the role specified, but will start up even if it must assume a different role.

This table describes startup behavior for the three policies.

**Table 1.3 Gateway Hub Startup Policy and Startup Behavior**

Startup Policy	Startup Behavior
none (default)	If no primary is running for the id, the hub starts as primary. Otherwise it starts as secondary.
primary	Same behavior as for none with the addition that the hub logs a warning if it starts as secondary.
secondary	If a primary is present, the hub starts as secondary. If there is no primary, the hub waits for up to one minute for a primary to start. If no primary starts in that time, the hub starts as primary and logs a warning.

## Configuring Remote Sites to Point to Your Multiple-Hub System

The regions' hub specifications only affect which hub processes their outgoing events. For incoming events, hubs process every event received regardless of their destination regions. For high availability, specify all of your hubs as gateway endpoints on your remote site. For any event, the remote site will send to only one endpoint.

This is a valid remote gateway specification pointing to our example site.

**Example 1.40 Remote Site Configuration Pointing to the Multiple-Hub Site**

```
<cache>
  <gateway-hub id="Some remote gateway hub ...
    <gateway id="Sitela" ...
      <gateway-endpoint id="SitelaLucy" host="lucy" port="22222"/>
      <gateway-endpoint id="SitelaRicky" host="ricky" port="22222"/>
      <gateway-endpoint id="SitelbLucy" host="lucy" port="22223"/>
      <gateway-endpoint id="SitelbRicky" host="ricky" port="22223"/>
    </gateway>
  </gateway-hub>
```

## Basic Configuration Steps

To implement a multiple-hub system, first determine the configuration of the connections between the multiple distributed system sites. Then, for each site, follow these steps:

1. Identify the VMs that will host your gateway hubs, including which machines they will run on the the addresses they will use for gateway communication.
2. Decide the gateway hub startup policy for each VM's hubs.
3. Decide which regions will point to which hubs.
4. Following your planned configuration, for each hub define the gateways to the remote sites, including:
  - ▶ Endpoints - point to every hub instance on the remote site
  - ▶ Queue attributes
5. Enable regions to communicate updates to the gateways, specifying:
  - ▶ enable gateway
  - ▶ hub ID
6. Start all host VMs in the distributed system at the same time. This allows the primaries to assume their roles before any secondaries default to primary.

For details on the basic configuration of gateway communication, see the information on multi-site configuration in the *GemFire Enterprise Developer's Guide*.

## Manual Start for Gateway Hubs - Early Access

You can now declare gateway hubs in your `cache.xml` without automatically starting them. Before this release, any gateway hub declared in your `cache.xml` would automatically start when the cache was created. This is still the default behavior. You can now disable the hub startup by setting the new `gateway-hub` attribute `manual-start` to `true` in the XML file. If manual start is enabled, you must then start the hub through the API using one of the `GatewayHub.start` methods.

This option allows you to use the normal XML for hub configuration and to control when your system starts communicating with remote sites. This is useful for involved startup procedures, such as those required when recovering from a crash, or for any situation in which you want to ensure that cache initiation finishes before remote site operations are allowed in.

## Write Behind Cache Listener Using Gateway Hubs- Early Access

You can now implement a write-behind cache listener in your gateway hubs. This is accomplished by creating multiple gateways in a single distributed system member and configuring one of them with no incoming port specification. The hub with no incoming port will only process events from inside the distributed system, allowing you to run the hub with write-only behavior. Without a port specified, the hub does not start a server socket to listen for incoming communication. It only receives events from inside its own distributed system, notifies its listeners, and sends the events to any configured remote sites.

To support the new multi-hub option, a new Cache API method `getGatewayHub(String id)` retrieves a gateway hub for a specific ID.

