GemFire Enterprise Architectural Overview

Release 5.0

GemStone Systems, Inc.

About this Guide

This guide describes the major services and functions in the GemStone® GemFire Enterprise product. It contains the following sections:

- Chapter 1, 'What is GemFire?' provides an overview of the GemFire Enterprise product, its positioning as a one-stop solution for distributed data caching, in-memory database querying, event-driven data stream monitoring, and MOM-like distributed messaging.
- Chapter 2, 'Topologies and Caching Architectures' discusses typical GemFire topologies and demonstrates each topology with a real-world example.
- Chapter 3, 'How Does GemFire Work?' opens the hood to present how GemFire distribution and data virtualization mechanisms work
- Chapter 4, 'Persistence and Overflow' discusses how GemFire memory-based data is synchronized with a database. how the in-memory data is automatically backed up to disk for recovery, and how memory management can overflow in-memory data to disk when memory capacity is exceeded
- Chapter 5, 'Transactions' describes GemFire's support for in-memory cache transactions, how these transactions can be executed in the context of a JTA transaction
- Chapter 6, 'Querying' describes the Object Query Language and how it is used to query GemFire regions
- Chapter 7, 'Data Availability and Fail-over' presents details of how GemFire provides resiliency in the face of catastrophic failure
- Chapter 8, 'GemFire Administration' outlines the tools and techniques for monitoring and tuning a GemFire distributed application
- Appendix A - GemFire API Code Examples

1 - What is GemFire?

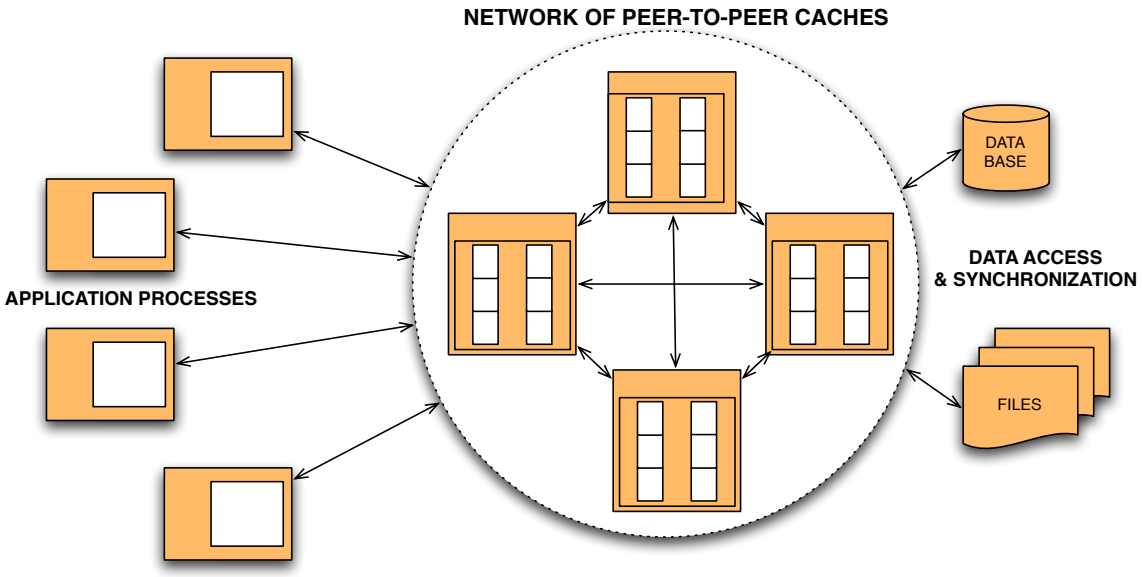
GemFire Enterprise is a high-performance, distributed operational data management infrastructure that sits between your clustered application processes and back-end data sources to provide very low-latency, predictable, high-throughput data sharing and event distribution.

GemFire harnesses the memory and disk resources across a network to form a real-time data fabric or grid. By primarily managing data in memory, GemFire enables extremely high-speed data sharing that turns a network of machines into a single logical data management unit – a data fabric.

GemFire is used for managing operational data – Unlike a traditional centralized disk-based database management system used for managing very large quantities of data, GemFire is a real-time data sharing facility specifically optimized for working with operational data needed by real-time applications – it is the "now" data, the fast-moving data shared across many processes and applications. It is a layer of abstraction in the middle tier that collocates frequently-used data with the application and works with back-end databases behind the scenes.

Distributed Data Caching

The most important characteristic of the GemFire Data Fabric is that it is fast – many times faster than the traditional disk-based database management system, because it is primarily main-memory based. Its engine harnesses the memory and disks across many clustered machines for unprecedented data access rates and scalability. It uses highly-concurrent main-memory data structures to avoid lock contention and a data distribution layer that avoids redundant message copying, and it uses native serialization and smart buffering to ensure messages move from node to node faster than what traditional messaging would provide.



It does this without compromising the availability or consistency of data – a configurable policy dictates the number of redundant memory copies for various data types, storing data synchronously or asynchronously on disk and using a variety of failure detection models built into the distribution system to ensure data correctness.

Key Database Semantics are Retained

Simple distributed caching solutions provide caching of serialized objects – simple key-value pairs managed in Hashmaps that can be replicated to your cluster nodes. GemFire, provides support for multiple data models across multiple popular languages – data can be managed as Java or C++ objects natively, and GemFire offers an XML:DB compatible API to store native XML documents.

Similar to a database management system, distributed data in GemFire can be managed in transactions, queried, and persistently stored and recovered from disk.

Unlike a relational database management system, where all updates are persisted and transactional in nature (ACID), GemFire relaxes the constraints allowing applications to control when and for what kind of data you need total ACID characteristics. For instance, a very high-performance financial services application trying to get price updates distributed can take advantage of what is most important - the distribution latency; there is no need for transactional isolation.

The end result is a data management system that spends fewer CPU cycles for managing data and offering higher performance.

Continuous Analytics and Reliable Event Distribution

With data in the fabric being updated rapidly by many processes and external data sources, it is important for real-time applications to be notified when events of interest are being generated in the fabric. Something a messaging platform is quite suited to do. GemFire data fabric takes this to the next level – applications can now register complex patterns of interest expressed through queries: queries that are continuously evaluated as new data streams into the data fabric. Unlike a database where queries are executed on resident data, the data fabric can be configured to be active – it instantaneously calculates how a query result has been impacted and routes the result set “delta” to distributed client processes.

When using a messaging platform, application developers expect reliable and guaranteed publish-subscribe semantics. The system has knowledge about active or durable subscribers and provides different levels of message delivery guarantees to subscribers. The GemFire data fabric layers these messaging features on top of what looks like a database to the developer.

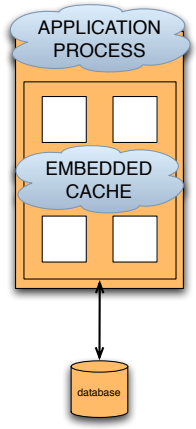
Unlike traditional messaging where applications have to deal with constructing piecemeal messages,, incorporating contextual information in messages, managing data consistency across publishers and subscribers, GemFire enables a more intuitive approach - one where applications simply deal with an object data model. They subscribe to portions of the data model and publishers make updates to the business objects or relationships. Subscribers are simply notified on the changes to the underlying distributed database.

2 - Topologies and Caching Architectures

Given the range of possible problems that a middle-tier data management system needs to address, a viable product must provide the components for architecting a system flexibly. GemFire provides many options with regards to where and how the cached data gets managed. To satisfy the networking, performance and data volume requirements, an architect can assemble the appropriate caching architecture from peer-to-peer, client-server and WAN components.

Peer-to-Peer Topology

In a peer-to-peer distributed system, applications can use GemFire 'mirroring' to replicate the data to other nodes or to partition large volumes of data across multiple nodes.



Peer-to-Peer Scenario: Embedded cache

In this cache configuration, the cache is maintained locally in each peer application and shares the cache space with the application memory space. Given the proximity of the data to the application, this configuration offers the best performance when the cache hit rate is high. Multiple such embedded caches can be linked together to form a distributed peer-to-peer network.

When using the embedded cache, applications configure data regions either using an XML-based cache configuration file or through runtime invocation of the GemFire API. The behavior of the cache and the patterns of distribution reflect the cache configuration options chosen by one of these methods. For example, a data region configured as a 'mirror' will replicate all data and events on the region across the distributed system when the member joins. This 'mirror' acts as a magnet – all operations on the distributed data region are automatically replicated on it, resulting in a union of all entries in that region across the distributed system.

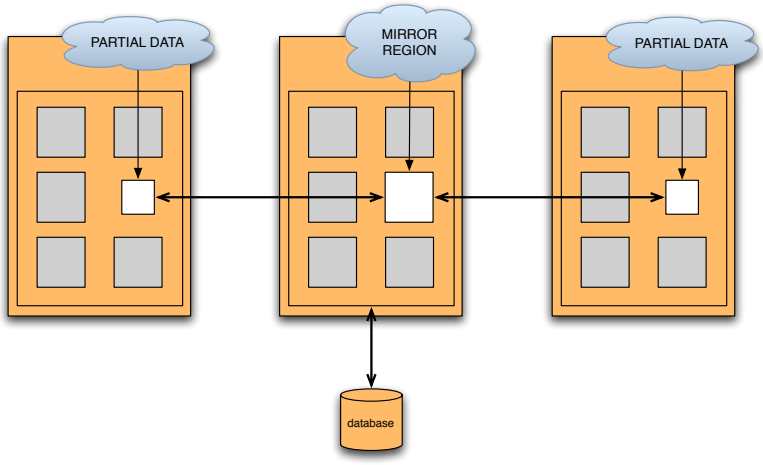


Figure: Aggregating data with a mirror region

A 'mirrored' region serves two primary purposes:

- It provides instantaneous access to data for READ_MOSTLY applications
- It provides recovery capability for other members in the system by replicating all the members' region values

Configuring a region to be a 'mirror' is a straightforward declaration in the member's cache configuration file. The following declaration creates a region named 'items' and establishes it as a mirror of all other 'items' regions in the distributed system:

```
<cache>
  <vm-root-region name='items'>
    <region-attributes mirror-type='keys-values'>
      </region-attributes>
  </vm-root-region>
</cache>
```

If a member cache with a 'mirrored' data region is created (newly created or recreated upon restart), GemFire automatically initiates an 'initial image fetch' operation – recovery of the complete state of the region from other members. This operation becomes an optimal directed fetch if there is another 'mirror' available for the data region. Otherwise, each member supplies the subset of entries it is managing.

Peer-to-Peer Scenario: Partitioned cache

Addressable memory on a given machine ranges from as little as 2GB (Windows) to 4GB (depending on the flavor of Linux) and actual allocation is limited to 1.5GB to 3.5GB in 32-bit operating environments. In addition to this limitation, if the cache is collocated with the application process, then the cache has to share this addressable space with the application, further limiting the maximum size of the cache. To manage data much larger than the single process space limit or the capacity of a single machine, GemFire supports a model for automatically partitioning the data across many processes and nodes. It provides this functionality through highly-available, concurrent, scalable distributed data structures.

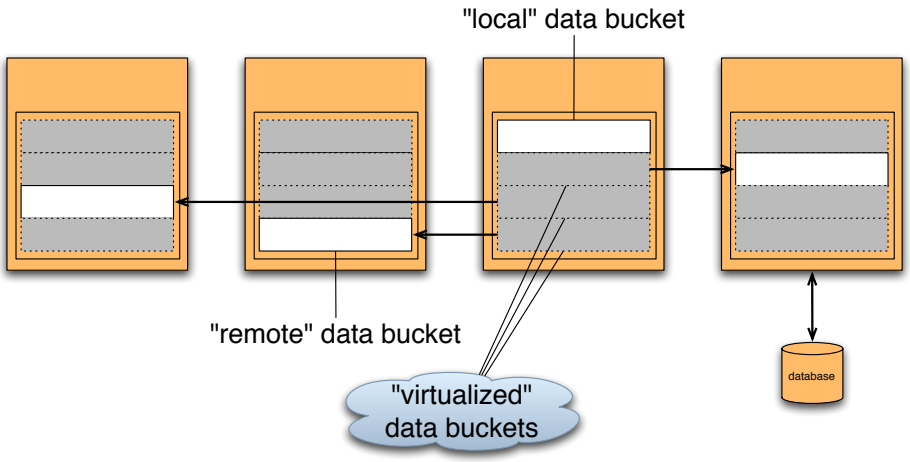


Figure: Partitioning data across many nodes

In a 'partitioned cache', all of the cache instances have unique regions (except for mirrored regions), and the peer-to-peer model can scale to over 100 cache instances holding tens to hundreds of gigabytes of data in main memory. Applications simply operate on the data locally and behind the scenes and GemFire manages the data across the data partitions of the distributed system while ensuring that data access is at most a single network hop. Region configurations control the memory management and redundancy behavior of these partitioned regions. The following declaration creates a region that is part of a larger partitioned region, and it indicates that all data will be dynamically put in 'buckets' that are determined by GemFire hashing algorithms. These hashed 'buckets' are spread across the available nodes (the default is 113 buckets, but it is configurable). The declaration also indicates that GemFire will replicate it and maintain the copy in a different node.

```
<cache>
  <vm-root-region name='items'>
    <region-attributes>
      <partition-attributes redundant-copies='1'>
      </partition-attributes>
    </region-attributes>
  </vm-root-region>
</cache>
```

Example: Declaring a partitioned region

Configuring the required level of redundancy automatically configures the error handling: when a node holding a partition fails, the data regions held in it are redistributed to other nodes to ensure that the desired

redundancy levels are maintained (with no duplication of an entry on any given node). Further, node failures will cause client requests to be redirected to a backup node automatically.

New members can be dynamically added (or removed) to increase memory capacity as the data volume managed in the data region increases without impacting any deployed applications. Thus, the GemFire system is very useful in applications where operational data size can grow to unknown size and/or the rate of data change is high.

Scaling a peer-to-peer system

A peer-to-peer system offers low latency, one-hop data distribution, dynamic discovery, and transparency of location. In this arrangement, members maintain direct, socket-to-socket connections to one another. When the number of members grows, the number of connections grows exponentially. This limits the scalability of an IP-multicast system to a few score members. To make the system more scalable, GemFire offers a reliable multicast (UDP) for scalable distribution of messages between data regions. When regions have this option enabled, the system may contain tens or hundreds of thousands of members without significant performance degradation.

Peer-to-peer distribution is useful for replicating data among servers, guaranteeing that the data is always available to clients. The next section outlines how a client-server topology uses a peer-to-peer architecture to replicate data in a server tier and how clients can use GemFire's load-balancing policies to maximize server performance.

Client-Server Topology

Client-server caching is a deployment model that allows a large number of caches in any number of nodes to be connected to one another in a client-server relationship. In this configuration, client caches—GemFire caches at the outer edge of the hierarchy, communicate with server caches on back-end servers. Server caches can in turn be clients of other server caches or can replicate their data by enlisting server peers as mirrors. This is a federated approach to caching data for a very large number of consumers and is a highly-scalable topology.

Applications use the client-server topology to host the data in one or more GemFire 'cache servers'. These cache servers are configured via XML files to host the data regions and to replicate one another's data using the peer-to-peer topology; cache servers can explicitly specify data regions to be 'mirrors' for replicating server-tier data to each other. As the 'mirroring' attribute is configured at a data region level for each server, application deployments have the flexibility to manage how and where the data is replicated across servers. For instance, one data region could be configured to 'mirror' on each cache server, while, a second data region could be configured such that only a partial data set is distributed across cache servers and the server can fetch the missing data from a back-end database upon a miss.

Transparent data access

A client can fetch data from its server transparently and on demand: if a client requests data that is not in its local cache (a miss) the request is delegated to the server cache. A miss on the server will typically result in the data being fetched from either one of its peers, another server in another server tier, or the back-end database that it (or one of its peers) is connected to. This configuration is well suited for architectures where there are a large number of distributed application processes, each caching a facet of the data originating from one or more server caches. With directed, point-to-point cache communications between the clients and the servers, the traffic to the client caches is dramatically reduced. The server cache is typically deployed on a bigger machine and, by holding common data its cache, shields unnecessary traffic to the back-end data source. The servers can be part of a distributed system, connected to one another as peers, mirroring data with one another for high availability and for load balancing of client requests.

Client-server communications

Communication between a server and a client is based on a connection created by the client. This allows a client to connect to servers over a firewall. Client caches can register interest lists with a server that identify the subset of data entries that they are interested in and when these data entries change, the server will send updates to the interested clients. Server-to-client communication can be made asynchronous, if necessary, via a server queuing mechanism that allows for a configurable maximum limit on the number of entries in the queue. Events pushed into the queue can also be conflated, so that the client receives only the most recent value for a given region entry. This ensures that the client-side performance does not bottleneck the cache server and impede its ability to scale to a large number of clients.

A multi-threaded client will, by default, open a separate socket to the server for each thread that is communicating to the server. In return, the server must open a socket for each of the client sockets. As the number of clients grows, server performance can degrade due to the large number of sockets that it uses to communicate with the clients. One way to manage this is to configure the clients' threads to share a pool of sockets. This frees the server to communicate all client requests using the same, single client socket and greatly increases the scalability of the clients.

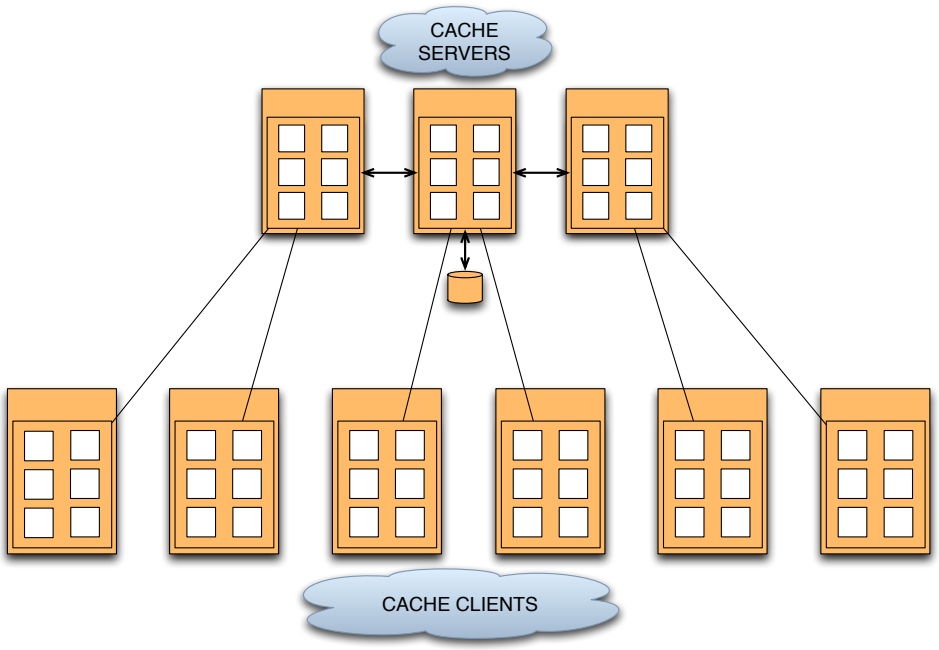


Figure: Cache server architecture

Clients connect to one or more cache servers using an XML configuration specified at the data region level. Each client's load-balancing policy determines how their TCP Connections to the servers are managed. Each client knows about all of the servers and is able to switch to a properly functioning server if one of the servers that it uses develops a problem.

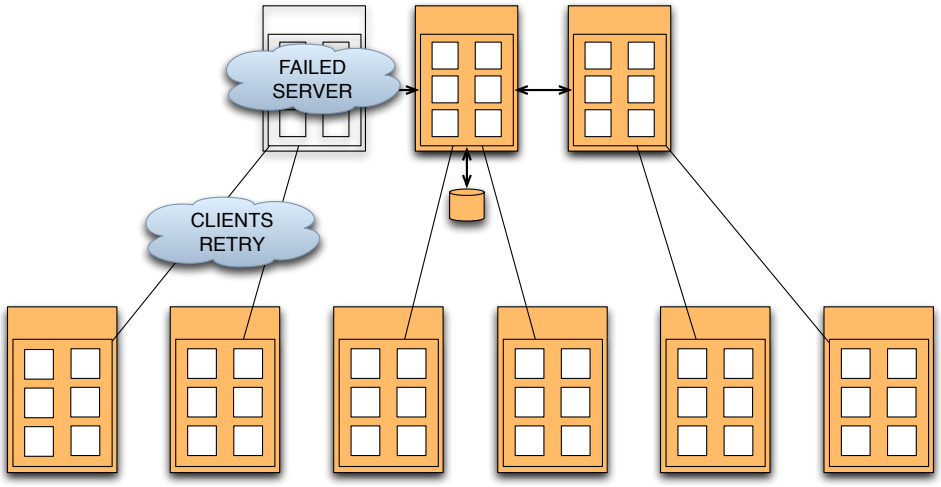


Figure: Failure of a cache server

In the event of an error or timeout in a server, the client will retry the message to the server based on the client's 'retryCount' that is specified in its configuration file. If the retryCount is exceeded, the server in question gets added to a failed server list, and the client selects another server to connect to. The servers in the failed server list are periodically pinged and if a server is deemed to be 'healthy' again, it is promoted

back to the healthy server list and the clients are load balanced to include the server. The time period between failed server pings is also configurable.

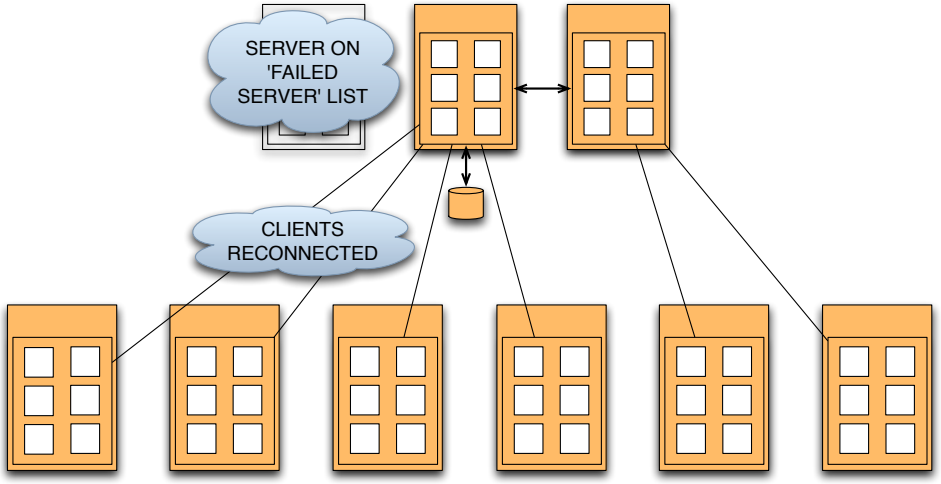


Figure: Clients reconnect to functioning server

Load balancing policies to distribute load across cache servers

Clients can also configure a load balancing policy to evenly distribute the load across cache servers, improving individual server responsiveness and resiliency. Three load balancing policies are supported:

Sticky: In this mode, a client picks the first server from the list of servers and establishes a connection to it. Once this connection has been established, every request from that particular 'client' cache is sent on that connection. If requests time out or produce exceptions, GemFire picks another server and then sends further requests to that server. This achieves a level of load balancing by redirecting requests away from servers that produce timeouts.

RoundRobin: In this mode, a client establishes connections to all the servers in the server list and then randomly picks a server for each given request. For the next request, it picks the next server in the list.

Random: In this mode, a client establishes connections to all the servers in the server list and then randomly picks a server for every request.

Data on demand

GemFire provides a simple set of plug-in interfaces for application developers to enable connectivity with remote data sources. To dynamically fetch data from an external data store, application developers plug in a 'CacheLoader' to load data from the external source into a GemFire cache. This loader is automatically invoked when an object lookup in the cache results in a miss. Upon loading data into the caller's local cache, GemFire distributes the new data to other cache nodes in accordance with the local cache's distribution policy and the remote cache's 'mirroring' policies. To synchronize changes to data in the cache with the external data store, the developer installs a write-through 'CacheWriter' in the cache that has close proximity to the data store. This writer synchronously writes changes to the data store before applying the change to the distributed cache.

Both plugins (the CacheLoader and CacheWriter) can be configured either through the GemFire cache XML configuration or dynamically with the GemFire caching APIs. Each loader or writer is associated with a single cache region, so the developer may install different loaders and writers in different cache regions. GemFire also offers flexibility on where cache loaders, writers and listeners are invoked. For instance, in a widely distributed environment, the data source may not be accessible from all nodes due to security constraints or the network topology. A cache miss on a cache that does not have access to the data source will automatically fetch the data from a remote data loader (usually in close proximity to the data source). Similarly, writers and listeners can be executed remotely. This loose coupling of applications to data sources allows new applications to scale across an enterprise without unnecessary costs associated with replicating data.

WAN Topology

Peer-to-peer clusters are subject to scalability problems due to the inherent tight coupling between the peers. These scalability problems are magnified if a data center has multiple clusters or if the data center sites are spread out geographically across a WAN. GemFire offers a novel model to address these topologies, ranging from a single peer-to-peer cluster all the way to reliable communications between data centers across the WAN. This model allows distributed systems to scale out in an unbounded and loosely-coupled fashion without loss of performance, reliability and data consistency. At the core of this architecture is a gateway hub for connecting to distributed systems at remote sites in a loosely-coupled fashion.

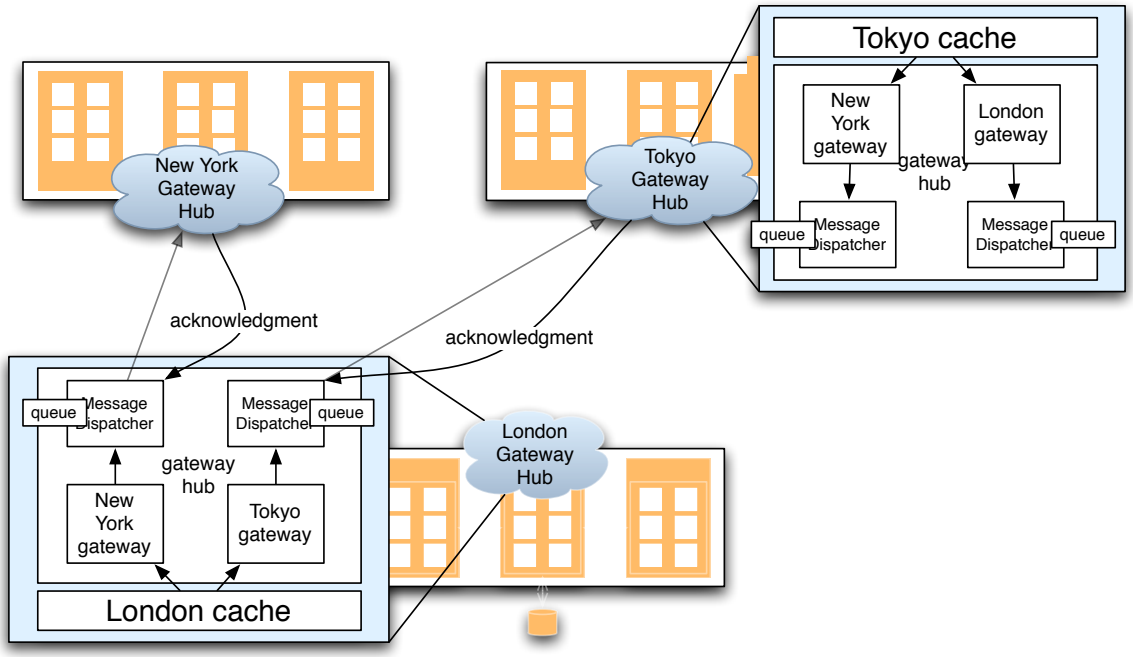


Figure: Gateway hub distributing data across a WAN

Each GemFire distributed system can assign a process as its gateway hub, which may contain multiple gateways that connect to other distributed systems. Updates that are made in one system can be propagated to another system via a queuing mechanism managed by each gateway. The receiving distributed system sends acknowledgments after the messages have been successfully processed at the other end. In this fashion, data consistency is ensured across the data centers. Messages in the gateway queue are processed in batches, which can be size-based or time-based.

The declarations below create a single gateway hub (server) that receives data from two locations through gateways (clients). It receives data from the two gateways through port '11111' and sends acknowledgments to each gateway through ports '22222' and '33333'.

```
<cache>
 <gateway-hub id='US' port='11111'>
  <gateway id='JP'>
    <gateway-endpoint id='JP' host='192.168.1.104' port='22222'/>
  </gateway>
  <gateway id='GB'>
    <gateway-endpoint id='GB' host='192.168.1.105' port='33333'/>
  </gateway>
 </gateway-hub>
</cache>
```

Example: Declaring a gateway hub and two gateways

GemFire can scale to thousands of nodes by having multiple distributed systems, where each distributed system hosts just a few cache servers and uses the WAN topology create a mega-cluster of distributed systems.

Error handling

When messages sent via a gateway are not processed correctly at the receiving end, those messages are resent or appropriate warnings or errors are logged, depending on the actual scenario in question. This flexible model allows several different topologies to be modeled in such a fashion that data can be distributed or replicated across multiple data centers so that single points of failure and data inconsistency issues are avoided.

Primary and secondary gateway hubs

Backup gateway hubs can be configured to handle automatic fail-over of the primary hub. As gateway hubs come online, each attempts to get a lock on a globally visible object. The hub that gets the lock becomes the primary. Any other hubs write a warning that they did not get the lock and idle, waiting on the lock. If the primary hub fails, the lock is released and the next hub in line gets the lock and becomes the new primary. The primary gateway hub maintains a persistent messaging queue that the secondary mirrors. This guarantees that when the secondary takes over from the primary, no data or messages will be lost.

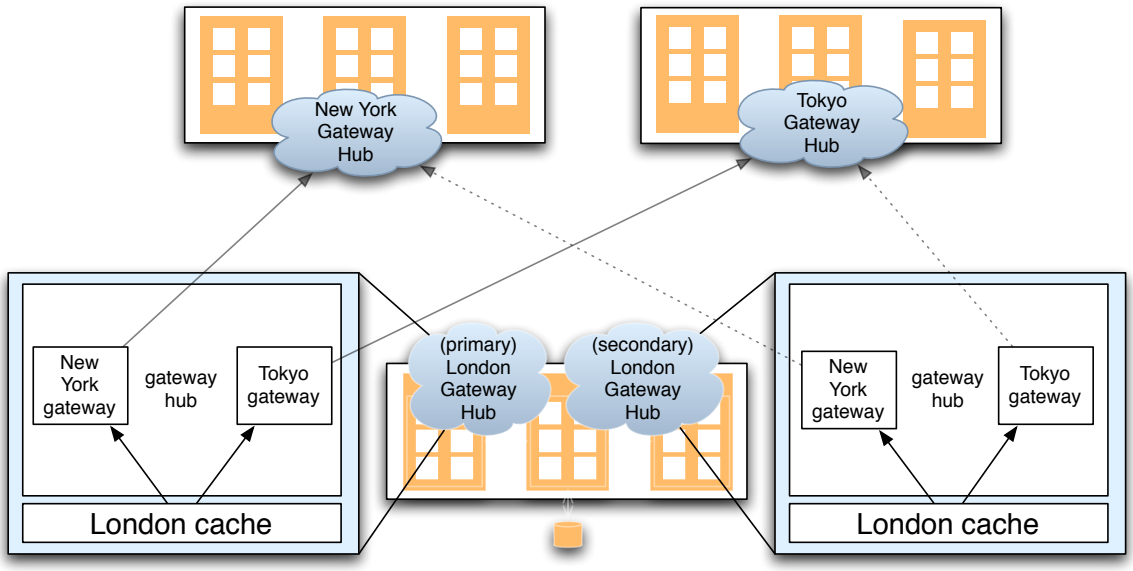


Figure: GemFire Secondary gateway hub guarantees availability

Case Studies

Here are some real-world examples of how these different topologies are being used in production by GemFire customers.

Case Study 1 – Trade Capture

A large brokerage firm has a trading application that must run on geographically-dispersed sites. All of the sites must have access to the same unified trade book in real-time.

1. Peer-to-peer. The brokerage firm created a foreign exchange trade book in a distributed system using two peer caches. The trade book is comprised of several cache regions to store various trade types. Both caches mirror each other to provide failo-ver and to load balance clients. One of the caches is made persistent so that the trade book can be recovered upon failure of both caches. The trade book is also synchronized with the relational database as a secondary storage.

2. WAN. Because of the high network latency over WAN that separates the geographical sites (Tokyo, London, and New York), the peer-to-peer system is duplicated on each site. Each site uses the WAN gateways to connect to the two other sites to fully replicate the trade book. There are three important benefits to this topology:

   a. WAN gateways guarantee data consistency and cache coherency over slow network. The trade book is now completely and globally synchronized.

   b. Each site runs at their own speed. Without this model, the high network latency over WAN will slow down the entire sites.

   c. Each site has two other sites to fail-over, effectively creating a perpetual distributed system.

3. Client-server. A client connects to the closest distributed system and lists the others as secondary fail-over distributed systems. For example, all clients in Asia use the Tokyo site as their primary distributed system. The London and New York sites are used as secondary. If the entire Tokyo site fails, then the clients are automatically connected to London or New York. Note that each site has two mirrored peer caches.

Benefits

This case study enabled the brokerage firm to create a single, unified trade book for all traders throughout the world. The trade book is updated at memory speeds in each site and delayed only by the WAN latency from one site to another. The overall performance increase of 10 folds has been achieved.

Case Study 2 – Reference and Historical Data

Upon successful deployment of the Trade Capture application, the same brokerage firm decided to make their reference data globally available. As in the Trade Capture scenario, all three system topologies were employed with some additional GemFire features.

1. Peer-to-peer. Because of the large amount of data, the data had to be partitioned into several caches. Using 64-bit platforms, they were able to load all their data from their relational database into a single distributed system. By making mirrored caches persistent, the system not only gained automatic disaster recovery but also its startup time was drastically reduced. Without GemFire persistence, the previous system needed to pre-load data from the relational database before the start of the day. Even with partial data, this pre-load process used to take more than an hour. With GemFire persistence, it now takes less than 2 minutes to pre-load the entire dataset. All of the caches are also synchronized with the relational database.

2. Client-server. The clients are configured exactly the same as in the Trade Capture system, except that each client is configured with a local cache for storing some of the frequently used data. Each client also has an option to persist their local cache such that they could work offline. Upon returning to online, the local cache is automatically synchronized with the latest data from the system.

3. WAN. Each site is given their own distributed system. For this use case, not all data needed to be shared. It turned out about less than a quarter of data is shared. The shared data are mirrored and the rest are made local to each site. Due to the static nature of the reference and historical data, only a very few updates were made throughout the day. Therefore the WAN latency has no bearing on the system performance.

Benefits

The 'Reference and Historical Data' use case is a classic example of how the distributed cache is used. It can hold enormous amounts of data – hundreds of gigabytes – that needs to be accessed at a very high speeds. This brokerage firm had each client querying data directly from their relational database. This traditional approach took a few seconds to perform a simple reference data lookup and several minutes to query historical data. With GemFire in place, the brokerage firm is now enjoying sub-millisecond reference data lookup and millisecond historical data retrieval.

Case Study 3 – Market Data

Market data distribution is typically achieved using a publish-subscribe messaging system that is either home-grown or commercially available. To remove the initial subscription delay, a market data system usually employs a cache to hold the last data updates. A client simply subscribes to message topics and thereafter receives a complete set of initial data along with real-time updates. GemFire takes this publish-subscribe model to the next level as this use case shows.

A securities firm has multiple market data feeds that they pump into their trading system using a leading messaging product. One of the feeds carries their own internal market data. All clients listen the message bus for topics of interest, such as security prices. There were three flaws to this firm's market data system.

1. Their messaging software cannot guarantee message delivery. Its 'Certified Messages' provides guaranteed message services; however, because of the file-based store-and-forward message delivery mechanism it is unable to keep up with the high data update rates that are inherent in market data systems. This firm continued to use the software and frequently experienced message drops and trade inconsistencies that ultimately led to incorrect positions. Each incident caused the firm to lose a significant amount of revenue as it missed market opportunities. Furthermore, there are many late-night system maintenance and corrections that the support engineers had to face.

2. The market data caching services that the firm used has no concept of fail-over. If the caching services fail, there is at least 10 minutes of a blackout period for many of the users.

3. There was no facility in place to search market data. The clients had to subscribe to the interested topics and manually filter the messages. This was a very time-consuming and error-prone task. As such, there was hardly any search capability built into client applications.

At first, this firm was reluctant to switch to GemFire for messaging because of the already established messaging software infrastructure. To test the waters, they decided to capture their messages in GemFire and serve them to GemFire-enabled clients. This alone solved the second and third issues above. However, it did not solve the first issue, the most important of three.

1. Peer-to-peer. The market data feeds are directly pumped into GemFire peer caches. Simple feed handlers were created and embedded into each peer. The peer caches are configured to provide guaranteed message delivery services and to conflate messages to prevent a slow client from holding up the entire distributed system.

2. Client-server. Each client subscribes to interested topics (keys in GemFire) using regular expressions. For search capability, each client is now able to submit ad-hoc queries using GemFire's built-in OQL (Object Query Language) to the distributed system for quick search.

3. WAN. For this application, they have created two distributed systems: one in the primary data center and the other in the backup data center.

Benefits

Using GemFire, this securities firm is now free of losing real-time messages. It is also able to perform ad-hoc queries opening up a slew of market data applications essential in algorithmic trading. Furthermore, there is almost no chance of experiencing a black out time due to cache failure. All these benefits were achieved without losing any system performance.

Case Study 4 – Risk Management

The Risk Management group of a top tier brokerage firm runs hundreds of liquidity and risk batch jobs at the end of the day to prepare for the next day's business. Some of the batch jobs are taking more than 40 minutes to complete. After studying the batch job business rules, GemStone discovered that most of the time is spent on querying data from their relational database and transforming the result sets into business objects. So GemStone created a simple solution that cached all pertinent data in the form of business objects in GemFire.

1. Peer-to-peer. Since this is a non-real-time application, the Risk Management group decided to create just one distributed system with several peer caches holding partitioned data. Each peer is responsible for designated data types. Because each cache is pre-loaded independently and concurrently, the pre-load time is significantly reduced, and more importantly, the business objects are readily available to all batch jobs. They no longer needed to load and transform data from the database. Furthermore, each peer cache has been fitted with a command execution mechanism such that they can execute business rules (or tasks) independently. The clients typically send tasks to the caches, which execute them on behalf of the clients. This allowed the tasks to be sent to where the data resides, rather than retrieving large amounts of data from clients.

2. Client-server. Because there are only a few clients that execute batch jobs, the WAN model was unnecessary. Instead, a generic client peer is used to send business rules (tasks) to the peer cache that holds interested data. In many instances, a single task spans multiple peer caches that hold orthogonal data, achieving grid-like parallel processing. This further significantly reduced the overall processing time.

3. WAN. The Risk Management group allowed other groups to tap into their distributed system for sharing data. It used GemFire role-based security policy to permit only read access to select data. The groups are intentionally forced to couple their own distributed system to avoid having heavy client loads. This not only lifts the client load-balancing responsibility from the Risk Management group, but also the provides a complete data-access independence to the other groups.

Benefits

With GemFire in place, a batch job that used to take 40 minutes is reduced to less than one minute to complete. The overall performance gain is 40-fold. Furthermore, they are now able to share business objects to other vertical applications at memory speeds. This was unachievable before GemFire and is creating new intra-day applications that are poised to generate extra revenues for the firm.

Case Study 5 – Program and Algorithmic Trading

A brokerage firm was in need of increasing the performance of their program trading system. They have been enjoying a steady increase in trade volume over several months but it was also hurting the system performance despite the hardware upgrades. Furthermore, they needed a way to revamp their algorithmic trading piece of the system to generate more complex real-time events to meet the rapidly changing market business requirements.

After hitting the wall tuning their relational database, they have decided to integrate GemFire into their system. This meant overhauling the entire program trading system, primarily because of the business rules which are embedded in the stored procedures. However, after about two weeks of data modeling based on the existing system data access patterns, GemFire was ready to be integrated.

1. Peer-to-peer. All of the trading activities occur around their order book, which needed to be accessed at a very high speed by all of the clients. Furthermore, the order book must be fail-safe. It must allow concurrent access, and more importantly, resilient from failure. Satisfying these requirements required the master-slave pattern, which provides fast transactional access to the order book and automatic system fail-over. The master has an embedded peer cache that has the sole control of the order book, which is comprised of multiple cache regions organized by data types. A slave also has an embedded peer cache, which mirrors the entire order book. There may be one or more slaves running as hot backups to the master. The slaves are restricted from interacting with external sources such as exchanges and DMA (Direct Market Access) but are fully autonomous in executing master-independent client commands (requests). If the master crashes, the slaves hold an automated election process that determines a new master among themselves. The newly elected master takes control of the order book and the system runs uninterrupted. The clients also continue to run seamlessly with no disruption and no awareness of the fail-over.

2. Client-server. All of the clients are configured to use a local cache that is automatically updated with only the interested data from the order book. For transaction services, the clients submit commands to the transaction queue (a cache region) for the master and slaves to consume. This queue works much like a message queue, which is mirrored to all slaves. The master's responsibility is to dispatch and execute the commands in the queue. A command is made up of a set of tasks (or business rules) which can be transactional and are automatically executed by the master. A task maybe as simple as an order entry or as complex as an algorithmic trading strategy.

3. WAN. Two distributed systems are created. One in the primary data center and the other in the backup data center. The order book is completely replicated to the backup data center.

Benefits

With GemFire in place, the brokerage firm today enjoys a 10-fold increase in performance, the master-to-slave auto-failover facility, and the flexibility of the command/task pattern for performing algorithmic trading driven by the end-users. It also provides open scalability of holding multiple masters for load-balancing the clients and trading activities.

3 - How Does GemFire Work?

Typically, a GemFire distributed system consists of any number of members that are connected to one another in a peer-to-peer fashion, such that each member is aware of the availability of every other member at any time. Since each distributed system is identified by an IP address and port, multiple disjoint distributed systems can reside on the same network of machines and use the same data sources.

A member uses the distributed system address and port specification to discover all the members connected to it. By default, GemFire uses IP multicasting for the discovery process, while all member-to-member communication is based on TCP. For deployment environments where internal network policies disallow the use of multicast or where the network spans multiple subnets, GemFire provides an alternative approach through the use of lightweight 'locator' servers that keep track of all member connections. New members are configured to consult a locator process when joining the distributed system and are able to establish the same direct socket-to-socket TCP connections that result from the IP multicasting discovery process. Once connected through either multicasting or a locator service, each member has a unique name that allows administrators to more easily monitor each member's data and performance.

Members can join and leave at any time without impacting other member nodes. For instance, heavy loads on a clustered 'stateful' application can be easily accommodated by adding more nodes dynamically. Similarly, a reduction in the application load would allow administrators to remove nodes without affecting the remaining members.

Data Distribution

Each member typically creates one or more cache data 'regions'. Each region is created with configuration attributes that control where the data is managed (the physical location), the region's data distribution characteristics, and its memory management and data consistency models.

Data distribution between members is intelligent as each member has enough topological information to know which members potentially share the data it is managing. The\is enables the distribution layer to route data, operations, and messages only to the right nodes.

By default, GemFire provides a peer-to-peer distribution model where each system member is aware of every other member. GemFire offers a choice in the transport layer - TCP/IP or Reliable Multicast (UDP). At a region level, multicast communication can be turned on or off based on local network policies and other considerations. Cache distribution and consistency policies are configured for on a region-by-region basis.

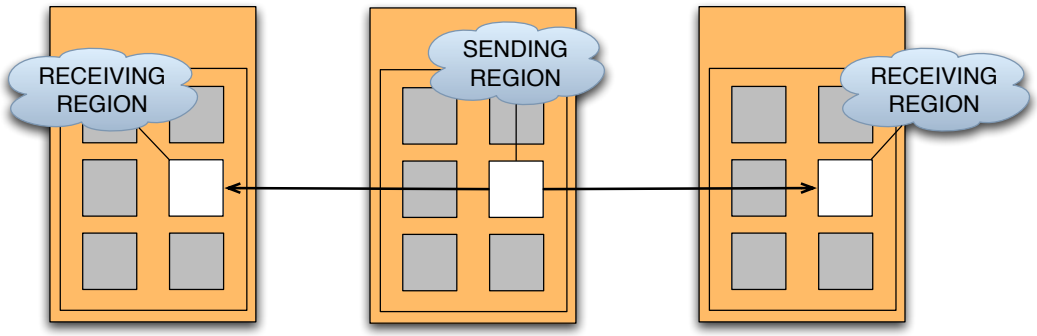


Figure: Data distribution

GemFire supports various distribution models, languages and consistency models for applications to choose from.

For replicating data across many cache instances, GemFire offers the following options:

- **Replication on demand**: In this model, the data resides only in the member region that originally created it. Data is distributed to other members' regions only when the other members request the data. The data is lazily pulled from the originator to the requesting member. Once the data arrives, it will automatically receive updates as long as the member region retains the key or maintains interest in the object.
- **Key replication**: This model can preserve network bandwidth and be used for low bandwidth networks because it makes the data available to all members by pushing the data entry key to the other members. Once a member has the key, it can dynamically 'get' the data when it needs it.
- **Total replication**: Both the data and its corresponding key are replicated.

Region Scope

Data is stored in the cache in cache 'regions'. Each region generally contains a particular type of data or a subset of data that logically belongs together. Each region has two key configuration parameters: its *scope* and its *mirror-type*. Scope controls whether or not the region will distribute its data to other member regions *of the same name*, and if it does distribute the data, how it guarantees that the message and data arrived on the receiving end. Scope comes in four flavors:

- Local - No distribution
- Distributed-no-ack - Send the data to the first member and send the data to the next member without waiting for any acknowledgment from the first
- Distributed-ack - Send the data and wait for an acknowledgment from the receiver before sending the data to the next member
- Global - Before distributing any data, get a distributed lock on every entry that will be updated in the other members. Once the lock is had, distribute the data. When all data is securely in place, release all of the locks.

Scope controls the sending of the data.

When a member notices incoming data (it's listening on a socket buffer), it scans the incoming message to determine how it should respond to the sender.

Synchronous communication without acknowledgment (distributed-no-ack scope)

Applications that do not have very strict consistency requirements and have very low latency requirements should use synchronous communication model without acknowledgments to synchronize data across cache nodes. This is the default distribution model and provides the lowest response time and highest throughput. Though the communication is out-of-band, the sender cache instance reduces the probability of data conflicts by attempting to dispatch messages as quickly as possible.

Synchronous communication with acknowledgment (distributed-ack scope)

Regions can also be configured to do synchronous messaging with acknowledgments from other cache members. With this configuration, the control returns back to the application sender only after a receiving cache has acknowledged acceptance of the data. Each message is sent synchronously in this way until all receivers have been sent the data and each has acknowledged storing it.

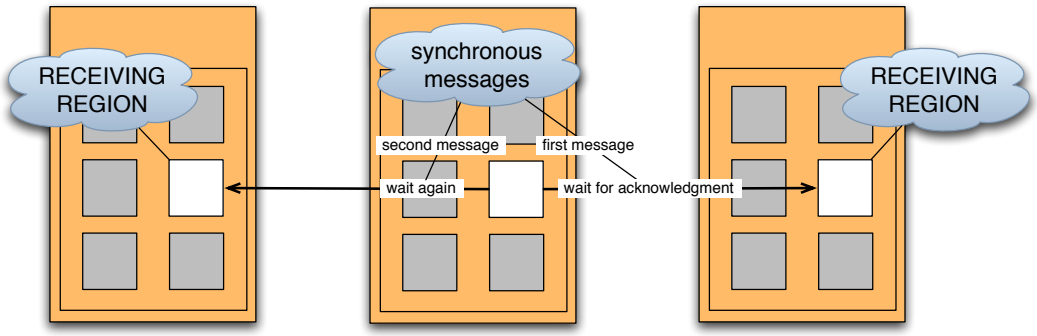


Figure: Synchronous distribution

There is a special form of this synchronous messaging that causes the receiver to send the acknowledgment as soon as it receives the message but before it stores the incoming data in its local cache. This 'early-ack' behavior improves performance when the time it takes to store the data in the receiver is less than or equal to the time it takes for the sender to dispatch the message. If the receiver's processing time is greater than the time it takes the sender to dispatch the message, there will be no performance improvement because the sender's socket buffer will eventually fill up and the sender will be slowed down to the receiver's speed.

Synchronous communication with distributed global locking (global scope)

Finally, for pessimistic application scenarios, global locks can first be obtained before sending updates to other cache members. A distributed lock service manages acquiring, releasing and timing out locks.

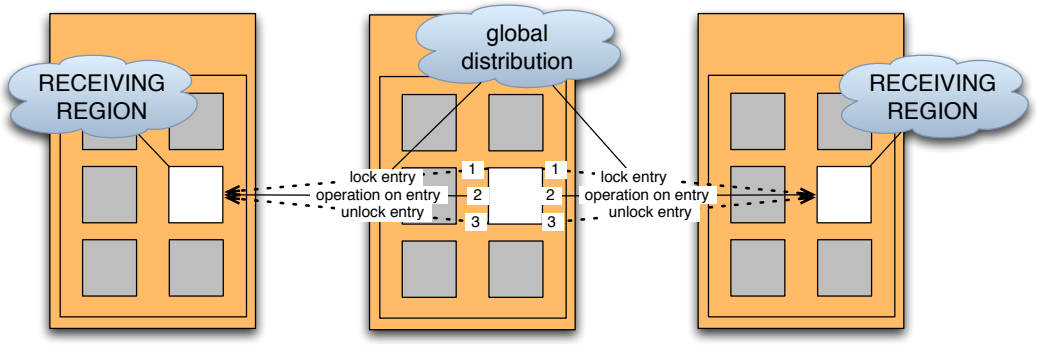


Figure: Distributed locking of region entries by a global region

Any region can be configured to use global locks behind the scenes through simple configuration. Applications can also explicitly request locks on cached objects if they want to prevent dirty reads on objects replicated across many nodes.

Local regions

Why is a region that doesn't distribute data useful? Why not just store POJOs in a Map in the JVM heap? GemFire offers several features that make using a 'local' (not distributed) region useful:

- Persistence - Data in a local region can be automatically persisted to disk. In the event of an application crash, the data is automatically restored to the region when the application is relaunched.
- Querying - Data in a local region can be queried using OQL (Object Query Language)

- Transactions - Operations on data in a local region can be wrapped in GemFire cache transactions to ensure the integrity of the data.

Region Mirroring

GemFire distributed regions are groups of data that have their own distribution, persistence, and data management policies. The data in the sending region is serialized into the socket buffer of each receiving member and each member is sent the data synchronously. In the case of a receiver that slow down this synchronous behavior, the sender can treat the slow receiver differently: by switching to an synchronous queuing of the messages to this 'slow receiver'.

When the region on the receiving end hears the data arrive in its socket buffer, it checks the header of the data to determine how to respond to the receiver. Receiver acknowledgments are based on the sender's scope: the sender message contains a description of the scope of the sending region in the message header.

After determining when to acknowledge the message, the receiver processes the newly-arrived data according to its 'data-policy'. It can:

- Store the data only if it already has an older version (mirror-type= 'none')
- Store the key to the data and ignore the newly-arrived value (mirror-type='keys')
- Store both the key and the value (mirror-type='keys-values'

Each distributed region in the distributed system has its own designated sending and receiving behavior and is the result of partitioning the data into groups and assigning each group of data its own distribution characteristics according to the application's requirements.

Keeping updates to selected data (mirroring off)

If one member's region has been designed to hold a subset of the other members' region data, the region will:

- load its data on initialization
- be configured with mirror-type='keys'

By loading the subset of data that it is designed to hold and by mirroring its data with 'mirror-type=keys', the region will only accept updates to data that it already has. Any new data sent by other members will be ignored.

Providing access to remote data (mirroring keys only)

Mirroring keys provides a region with access to all data in the other members, but saves memory by forcing to fetch the values on demand. When the local member requests the data from the region, the region will know that the values exists elsewhere in the cache by virtue of having the key. It then uses the key to perform a 'net search' of other members, fetches the data from another member, and stores it locally when it receives it. Once the value is stored locally, it will accept updates to the value as they arrive.

This scenario is useful when an application can't predict which data it will need on a given node, but once data is accessed on the node, it will often be needed again.

Keeping all data locally (mirroring keys and values)

A region with mirror-type set to 'keys-values' is a true mirror. It will duplication any region that has a full data set, or it will aggregate data from other regions into a full data set locally. It is critical for creating redundancy in server tiers and for applications that require immediate access to all data on a node.

Cache Configuration

Data management policies and distribution patterns are typically established by members' property files and XML cache configuration files. These configurations are remarkably easy to declare. This extract from the system property file indicates that the member will be joining the distributed system associated with mcast address 'goliath' and mcast port '10333'.

```
# in gemfire.properties file
mcast-address=goliath
mcast-port=10333
```

The next extract from the member's cache configuration file will cause the member to create two regions, one of which ('items') will distribute data and wait for acknowledgment from each member that it distributes to (scope='distributed-ack') and it will only accept updates to data that it already has (default mirror-type='none'). The other region ('offerings') will not wait for acknowledgment when it distributes data (it uses the default scope='distributed-no-ack') but will act as an aggregator of all of the 'offerings' entries in the distributed system caches (by declaring mirror-type' 'keys-values'):

```
<!-- in cache.xml file -->
<cache>
  <vm-root-region name='items'>
    <region-attributes scope='distributed-ack'>
    </region-attributes>
  </vm-root-region>
  <vm-root-region name='offerings'>
    <region-attributes mirror-type='keys-values'>
    </region-attributes>
  </vm-root-region>
</cache>
```

Example: Creating a cache and two regions declaratively

The same result can be had by using the GemFire programming API. When the application launches, it can configure the cache and its regions by setting system properties and region attributes programmatically:

```
// connect to the distributed system
Properties properties = new Properties();
properties.setProperty("mcast-address", "goliath");
properties.setProperty("mcast-port", 10333);
DistributedSystem system = DistributedSystem.connect(properties)

// create the cache (it will be used to create the regions)
cache = CacheFactory.create(system);

// create the distributed-ack items region
RegionAttributes regionAttributes = factory.createRegionAttributes();
regionAttributes.scope=Scope.DISTRIBUTED-ACK;
Region region = cache.createVMRegion("items", regionAttributes);

// create the mirror offerings region
regionAttributes = factory.createRegionAttributes();
regionAttributes.mirrorType=MirrorType.KEYS-VALUES;
region = cache.createVMRegion("offerings", regionAttributes);
```

*Example: Creating a cache and two regions programmatically*The GemFire data operations are equally simple: application developers 'put' data, 'get' data, 'remove' data, and 'listen' for actions and operations on the data.

```
// demonstrate region operations
Object key = "foo_key";
Object value = "foo_value";

region = cache.getRegion("items");
region.create(key, value);
value = region.get(key);
region.put(key, value);
region.invalidate(key);
region.destroy(key);
```

Example: GemFire cache operations

The simplicity of the API belies the power of the caching mechanisms. Persistence, transactions, overflow, event notifications, data on demand, fail-over, wide area distribution, continuous queries ... all are easy to configure and use because much of the cache behavior is transparent to the developer.

Cache Behavior

Automatic backup of region data

Three attributes ('persist-backup', 'disk-dirs' and 'disk-write-attributes') control how and where region data is backed up to disk and where data that overflows to disk is written. Up to four directories may be specified with the data 'striped' into files in the specified directories. This attribute defaults to use one directory: the user's current directory. (see the 'Persistence and Overflow' section for more details).

Distributed event notifications

Cache listeners can be used to provide asynchronous event notifications to any number of applications connected to a GemFire distributed system. Events on regions and region entries are automatically propagated to all members subscribing to the region. For instance, region events like adding, updating, deleting or invalidating an entry will be routed to all listeners registered with the region. Data regions can be configured to have multiple cache listeners to act upon cache events. Furthermore, the order of events can be preserved when cache operations are performed within the context of a transaction. Event notifications are also triggered when member regions leave or enter the distributed system, or when new regions are created. This enables application interdependencies to be modeled in SOA-like environments. Applications can also subscribe to or publish region events without caching the data associated with those events. GemFire can thus be used as a messaging layer, which sends/receives events to multiple cacheless clients, with regions being equivalent to message destinations (topics/queues). Unlike an enterprise messaging system, the programming model is very intuitive. Applications simply operate on the object model in the cache without having to worry about message format, message headers, payload, etc. The messaging layer in GemFire is designed with efficiency in mind. GemFire keeps enough topology information in each member cache to optimize inter-cache communications. GemFire Intelligent Messaging transmits messages to only those member caches that can process the message.

How the 'get' operation affects the local entry value

If a 'get' operation misses and retrieves an entry value from outside the local cache through a local cache loader or by fetching the data from another system member, it automatically attempts to 'put' the retrieved value into the local cache for future reference. This 'put' will invoke any existing cache writer and run the local capacity controller if one is configured. Either of these plugins' operation can, in turn, abort the attempt to store the value in the local cache (the successive 'put' operation). The original 'get' operation always bring the entry value to the calling application, but it is possible for the fetched data to remain unavailable in the local cache because the successive put was aborted.

Dealing with member failures

In a standard peer-to-peer environment, all members are considered equivalent. But applications might organize members in a primary/secondary relationship, such that when the primary fails, one of the secondaries can take over with zero impact to the availability of the application. To facilitate this, GemFire provides a health-monitoring API and a membership-callback API. Applications register a membership event listener and receive notifications when members join or exit the system. Member identity and an indication of whether they exited gracefully or ungracefully are supplied to the application through these callbacks.

So, how does the distributed system detect member failures? Failure in the distribution layer is identified at various levels: socket exceptions that indicate link or application failures, timeouts in acknowledgment-based message processing or timeouts/exceptions in 'heartbeat' processing. (Once a member joins the system, it sends out a 'heartbeat' at regular intervals through its connections to other members to describe its health to the others and monitors incoming heartbeats from other members for signs of their health.)

Preparing Data for Distribution

The standard implementation of Java's Serializable interface is very inefficient. By including more meta information than is often needed, it usually results in larger-than-necessary serialized objects and marshaling and unmarshaling methods that are unnecessarily time consuming. To enable more efficient transfer of binary data from one cache instance to another, GemFire allow three different types of data to be distributed:

- implementations of Java's Serializable interface
- byte arrays
- implementations of GemFire's own DataSerializable interface.

Assuming that the developer wants objects (not byte arrays) stored in the cache, it is recommended that the developer tailor the serialization of each type of object by implementing the DataSerializable interface in the object's class and eliminating the extraneous meta information from the resulting stream of bytes. When GemFire distributes the object, it constructs a header with information about the type of serialization used and serializes the object into the socket data stream. When the receiver detects the incoming data, it inspects the header to determine what type of deserialization to use and invokes the appropriate methods on the stream to reconstruct the object.

Using DataSerializable can have enormous impact on the performance of the distributed cache. I can reduce marshaling, distribution, and unmarshaling times, reduce the number of times garbage collection is invoked, minimize any paging in memory-constrained machines.

```
public class Pixel implements DataSerializable {
  private Point point;
  private RGB rgb;
// constructors and accessor methods ...
...
// DataSerializable methods
  public void toData(DataOutput arg0) throws IOException {
    arg0.writeInt(point.x);
    arg0.writeInt(point.y);
    ...
  }

  public void fromData(DataInput arg0) {
    point = new Point(arg0.readInt(), arg0.readInt());
    rgb = new RGB(arg0.readInt(), arg0.readInt(), arg0.readInt());
  }
}
```

Example: Defining a DataSerializable type

It is also possible to shorten the time spent instantiating a serialized object; using the GemFire Instantiator class to instantiate DataSerializable objects is much faster because it avoids the reflection mechanisms that are used during the process of unmarshaling Serializable objects.

```
public class Pixel implements DataSerializable {
  private Point point;
  private RGB rgb;

  static {
    Instantiator.register(new Instantiator(Pixel.class, (byte) 01) {
      public DataSerializable newInstance() {
        return new Pixel();
      }
    });
  }

// constructors and accessor methods ...
// creates an empty user whose data is filled in its fromData() method
  public Pixel() {}


// DataSerializable methods
  public void toData(DataOutput arg0) throws IOException {
    arg0.writeInt(point.x);
    arg0.writeInt(point.y);
    arg0.writeInt(rgb.red);
    arg0.writeInt(rgb.green);
    arg0.writeInt(rgb.blue);
  }

  public void fromData(DataInput arg0) {
    point = new Point(arg0.readInt(), arg0.readInt());
    rgb = new RGB(arg0.readInt(), arg0.readInt(), arg0.readInt());
  }
}
```

Example: Using an Instantiator

C++, C#, and .NET support and interoperability

Given that most IT environments are characterized by more than one application platform, it is imperative for a distributed data fabric to provide interfaces and interoperability across multiple languages. GemFire Enterprise offers APIs for C++ and .NET clients to access the distributed cache and perform the cache operations that are available to Java applications via the standard Java APIs.

Client applications can be developed in any of the .NET languages and use the GemFire java cache. The GemFire .NET Adaptor is a 'wrapper' for the GemFire Java API. .NET application simply include the 'wrapper' libraries and use the API using .NET naming conventions and syntax. Many common .NET data types are marshaled and unmarshaled to and from the Java cache without any additional developer effort.

To store and fetch user-defined .NET objects in the Java cache, the developer must provide Java implementations for these classes, use the provided tools to generate proxies based on the Java .class files, compile and link the .NET assembly, and include the assembly in the Visual Studio project. With these proxy libraries referencing these proxy libraries in the project, .NET objects can be instantiated, passed as arguments to the operations and declared as return types in the .NET/Java adaptor API,. All marshaling and unmarshaling is done by the adaptor API wrappers.

C++ environments have two options: use the .NET wrapper to build managed C++ clients that use the GemFire cache, or use GemFire Enterprise - C++ to provide native distributed caching for C++ applications. GemFire Enterprise - C++ caches objects in a native C++ format and eliminates the need for mapping Java objects to C++.

Scenario: Hooking Microsoft Office clients up to the GemFire cache

Since the GemFire .NET Adaptor provides proxies to the GemFire Java API classes, developing Microsoft Office applications that use the GemFire java cache is simple: just develop Office plugins in Visual Studio and invoke the cache operations from the .NET adaptor classes. All communications from .NET to Java is handled transparently by the .NET Adaptor.

Native XML Data Management Services

Besides managing objects, GemFire can also store XML documents in the distributed cache. GemFire implements the XML:DB Java APIs to enable access to XML document collections in the cache. XML:DB is a standards initiative that defines a simple Java API for access to XML repositories. The GemFire XML:DB implementation provides the XML:DB APIs for managing XML document collections, but extends this with the caching and distribution semantics described earlier. GemFire Enterprise also supports bi-directional XML/Java object transformation, which converts the documents into a form that can be used with the GemFire Java APIs and vice-versa. The system also includes a document-filtering pipeline that can validate or alter documents as they are put into or taken out of the cache. This allows documents to be validated, rejected, corrected, or transformed before entering or returning from the cache.

These APIs provide access to XML documents as W3C DOM, SAX events or as Strings. Applications use XPath-based queries to selectively fetch data within XML documents. GemFire extends W3C XPath semantics so that XPath expressions can be applied to an entire XML document collection at once. The querying implementation is highly optimized through a main-memory based indexing scheme and an efficient, compact memory representation for XML documents that retains the structure and relationships between various elements in a document.

Remote cache access using HTTP/SOAP and HTTP-direct

Access to the XML collections is also exposed as web services accessible via the SOAP 1.1 protocol. Published WSDL (Web Services Description Language) documents provide heterogeneous access to the cache from remote applications. GemFire can also accept XML documents delivered via HTTP requests The objects are transferred as the payload and a set of HTTP header properties define the cache operations. This feature provides an alternate efficient means (compared to SOAP over HTTP) to access information in the cache.

4 - Persistence and Overflow

Region data can be stored to disk for backup purposes (persistence) and to satisfy region capacity restrictions without destroying the cache-local data (overflow). These mechanisms use disk files to hold entry data. For overflow, only the entry values are copied to disk while keys remain in memory to facilitate dynamic retrieval of the disk-based values. For persistence, the entire data set (keys and values) are copied to disk to facilitate restoring the region data in case of a member failure. This persisted data outlives the VM where the region resides and can be used to initialize the region at creation. In contrast, overflow acts only as an extension of the region in memory and is deleted when the member exits the distributed system. Persistence and overflow can be used individually or in combination. The following sections describe the combinations.

Persistence

To optimize disk writes, the cached regions can be configured to either write all changes to disk synchronously during the operations, or asynchronously at regular intervals. The latter option should be used cautiously because if a member crashes during the interval when region values are accumulating for the disk write, the updated values will be lost when the member fails. It should only be used by applications that can tolerate incomplete recovery upon failure.

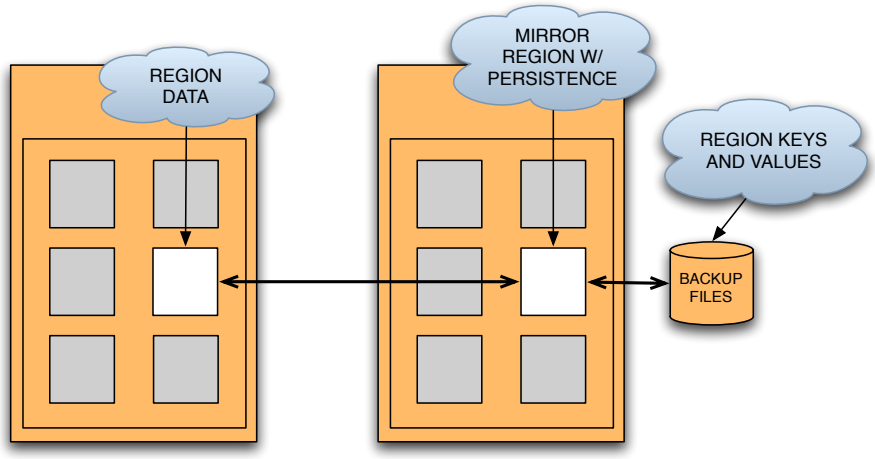


Figure: Using a persistent mirror to backup data

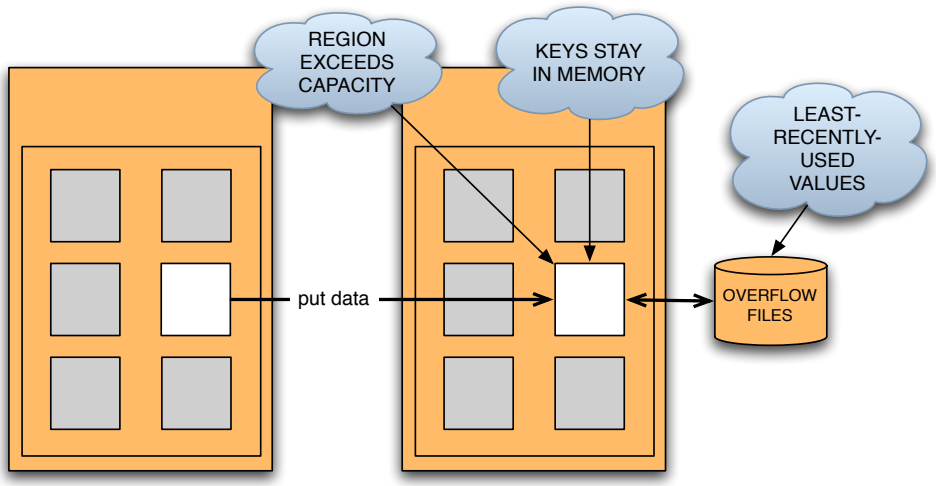
Persistence is specified as a region attribute by setting the 'persist-backup' region attribute to true.

```
<cache>
  <vm-root-region name='items'>
    <region-attributes persist-backup='true'>
      <disk-dirs>
        <disk-dir>vol1/items_backup</disk-dir>
        <disk-dir>vol2/items_backup</disk-dir>
        <disk-dir>vol3/items_backup</disk-dir>
        <disk-dir>vol4/items_backup</disk-dir>
      </disk-dirs>
      <disk-write-attributes>
        <asynchronous-writes time-interval='15'/>
      </disk-write-attributes>
    </region-attributes>
  </vm-root-region>
</cache>
```

Example: Declaring a persistent region and its asynchronous write behavior

Overflow

Overflow is implemented as an eviction option in the LRU capacity controllers and is specified in the cache configuration file by installing a capacity controller in a region with an eviction action of 'overflow-to-disk'.



Configuring a region for overflow requires a declaration among the region's region-attributes:

```
<cache>
  <vm-root-region name='items'>
    <region-attributes persist-backup='true'>
      <capacity-controller>
        <class-name>
          com.gemstone.gemfire.cache.util.MemLRUCapacityController
        </class-name>
        <parameter name='maximum-megabytes'>
          <string>50</string>
        </parameter>
        <parameter name='eviction-action'>
          <string>overflow-to-disk</string>
        </parameter>
      </capacity-controller>
      <disk-dirs>
       <disk-dir>vol1/items_overflow</disk-dir>
       <disk-dir>vol2/items_overflow</disk-dir>
       <disk-dir>vol3/items_overflow</disk-dir>
       <disk-dir>vol4/items_overflow</disk-dir>
     </disk-dirs>
    </region-attributes>
  </vm-root-region>
</cache>
```

Example: Declaring a CapacityController that overflows data to disk

Persistence and Overflow In Combination

Used together, persistence and overflow keep all region entries (keys and data values) on disk with only the most recently-used entries retained in memory. The removal of an entry value from memory due to overflow has no effect on the disk copy as all entries are already present there due to the persistence.

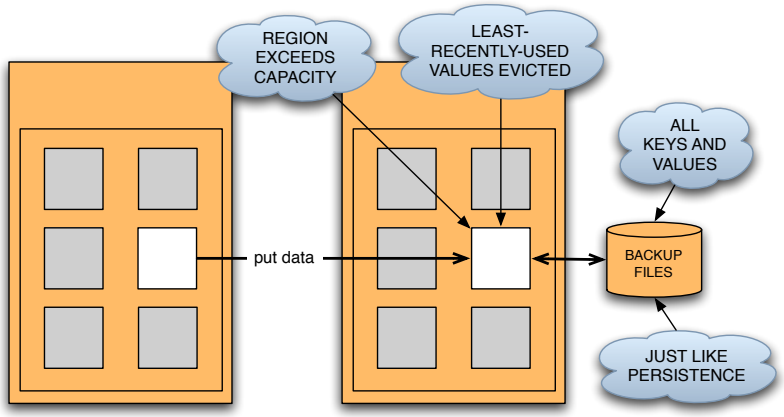


Figure: Combining persistence and overflow

Performance Considerations

While the performance of accessing data on disk has been shown to be quite high, there are a couple of considerations to take into account when using disk storage.

When an entry is overflowed, its value is written to disk, but the entry that held it and the entry's key remain in the region. This scheme allows the region to receive updated entry values, but also incurs some overhead in the VM: in addition to the size of the key data, each entry object itself occupies approximately 40 bytes of data.

When a mirrored backup region is restored, it is populated with the entry data that was found on disk. However, some of this data may be out of date with respect to the other members of the distributed system. To ensure that the mirrored region has the most recent data values, the cache requests the latest value of each piece of data from the other members of the distributed system. This operation could be quite expensive for a region that contains a lot of data.

Entry retrieval in overflow regions

For data value retrieval in a VM region with an overflow capacity controller installed, memory and disk are treated as the local cache. When a get is performed in a region with overflow enabled, memory and then disk are searched for the entry value. If the value is not found, the distributed system runs the standard operations for the retrieval of an entry whose value is missing from the local cache (loaders, remote cache requests, etc.)

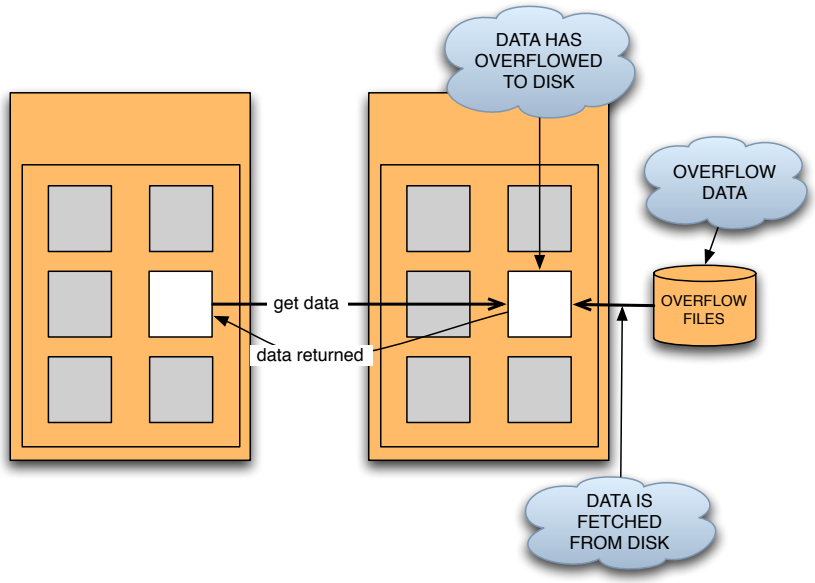


Figure: Retrieving data that has overflowed to disk

Entry retrieval in persistent regions

With persistence enabled and overflow disabled, the disk is not accessed for data value retrieval as it does not hold any values that are not also in memory. When you recover a persisted region, the region in memory is initialized only with the keys that are stored on disk; the values are copied from disk on demand.

5 - Transactions

GemFire provides support for two kinds of transactions:

• GemFire cache transactions - GemFire cache transactions control operations within the GemFire cache while the GemFire distributed system handles data distribution in the usual way. Use cache transactions to group the execution of cache operations and to gain the control offered by transactional commit and rollback.

• JTA global transactions - JTA global transactions provide another level of transactional control. JTA is a standard Java interface that can coordinate GemFire transactions and JDBC transactions under one umbrella. The GemFire Enterprise Distributed product includes an implementation of JTA, but GemFire can enlist in transactions controlled by any implementation of JTA. When GemFire runs in a J2EE container with its own built-in JTA implementation, GemFire also works with the container's manager.

GemFire Cache Transactions

Applications manage transactions on a per-cache basis. A transaction starts with a begin operation and continues with a series of operations, typically region operations. A commit, failed commit, or voluntary rollback ends the transaction. In GemFire cache transactions, the commit and rollback methods are directly controlled by the application.

A Java application can operate on the cache using multiple transactions. A transaction is associated with only one thread, and a thread can operate on only one transaction at a time. Child threads do not inherit existing transactions.

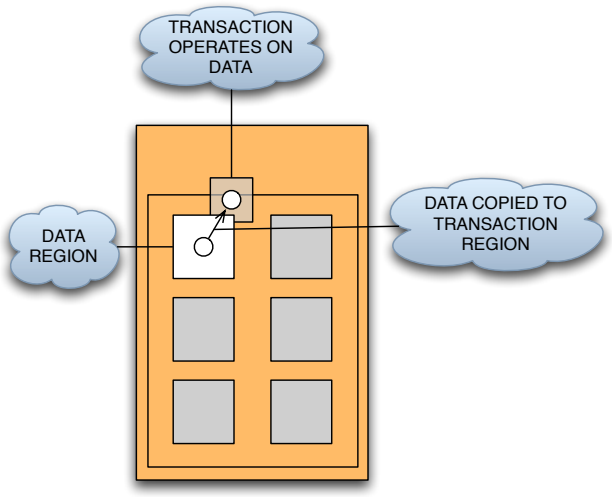


Figure: Private transaction region

Each transaction has its own private record of the changes it has made to the cache. In effect, the transaction has its own 'parallel and private' region that it creates and manages during the transaction. Operating on data in a transaction copies the data into the transaction's private region until the transaction is committed.

If there is no commit conflict during a commit, the transaction moves the changed data back into the original region. In this way, the transaction is isolated from changes being made concurrently to the cache.

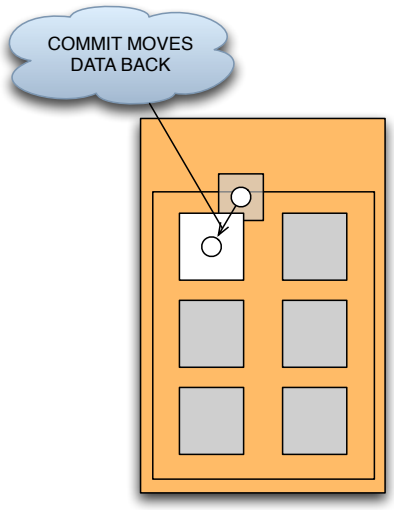


Figure: Committing data

The transactional view exists until the transaction completes. If the commit fails or the transaction is rolled back, the changes are dropped. If the commit succeeds, the changes recorded in the transactional view are merged into the cache. At any given time, the cache reflects the cumulative effect of all operations on the cache, including committed transactions and non-transactional operations.

Cache Transactions in the Distributed Cache Architecture

The data scope of a GemFire cache transaction's view is the cache. Because a transaction is managed on a per-cache basis, multiple regions in the cache can participate in a single transaction.

Transactional Data Sharing

Cache transactions are distributed in the same way that regions are distributed, that is, the changes they generate can be distributed. As always, the scope of the region determines the rules of the distribution.

Data distribution happens at commit time. Until then, changes made in a transaction are not distributed, even among threads in the same process.

Two threads cannot participate in the same transaction, even when they are in the same VM. A transaction in one VM cannot participate in a transaction in another VM.

ACID Properties of GemFire Cache Transactions

The data behavior of a transaction is defined by the ACID acronym: atomicity, consistency, isolation, and durability.

Atomicity means that either the transaction happens in its entirety, or it does not happen at all. Cache transactions are optimistic, so the check for conflicts happens at commit time. If a conflict exists, the commit fails, the transaction ends, and the cache does not change.

Consistency means that each transaction transforms the cache correctly. All changes are applied across all relevant regions to ensure a consistent state.

GemFire transactions support Read Committed isolation, which means that any read operation, whether transactional or not, reads only data that has been committed.

The changes made by GemFire transactions are durable in that once the transaction's changes have been applied, they cannot be undone.

Interaction Between Transactions and Region Scope

A cache transaction can involve operations on multiple regions, each of which can have different attributes. While a transaction and its operations are applied to the local VM, the resulting transaction state may be distributed to other VMs according to the attributes of each participating region.

A region's scope affects how data is distributed during the commit phase. Transactions are supported for these region scopes:

- Local - no distribution, handles transactional conflicts locally
- Distributed no ACK - handles transactional conflicts locally, less coordination between VMs
- Distributed ACK - handles transactional conflicts both locally and between VMs

All distributed regions of the same name in all caches must be configured with the same scope.

Local scope

Transactions on locally scoped regions have no distribution, but they do perform conflict checks in the local VM. You can have conflict between two threads when their transactions change the same entry. The first transaction to start the commit process "wins." The other transaction's commit fails with a conflict, and its changes are dropped.

You can also have a conflict when a non-transactional operation and a transaction modify the same entry. In that case, the transaction's commit fails.

Distributed-no-ack scope

If you need faster transactions with distributed regions, distributed-no-ack scope produces the best performance. This scope is most appropriate for applications with multiple writers that write to different keys, or applications with only one writer. For example, you can batch up groups of changes into transactions and do bulk distribution during large commits, thereby decreasing your overhead per change. The larger commit would make conflicts more likely, unless your application has a single writer. Then using the distributed-no-ack scope would be safe and give you maximum speed.

In distributed-no-ack regions, GemFire performs conflict checks in the local VM but not remote conflict checks. As in the local scope case, GemFire recognizes a conflict between two local threads and the second transaction that attempts to commit fails. If transactions on two different VMs conflict, however, they can overwrite each other if they both commit at the same time. The application probably would not know about the conflict in time to stop a commit, because the commit call can return while the changes are still in transit or in the process of being applied to the remote VMs.

Distributed-ack scope

Distributed-ack scope is designed to protect data consistency. This scope provides a much higher level of coordination among transactions in different VMs than distributed-no-ack.

When you use distributed-ack scope, when the commit call returns for a transaction that has only distributed-ack regions, you can be sure that the transaction's changes have already been sent and processed. Even after a commit operation is launched, the transaction continues to exist during the data distribution. The commit does not complete until the changes are made in the remote caches and the applications on other JVMs receive the callbacks to verify that those tasks are complete. Any cache and transaction call-

backs in the remote VM have been invoked. By now, any remote conflicts are reflected in the local region, and that causes the commit to fail.

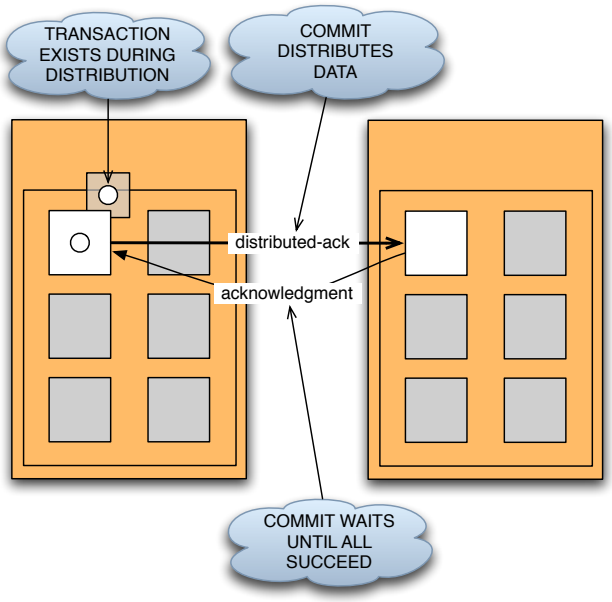


Figure: Distributed commits using distributed-ack scope

This scope is most appropriate for applications with simultaneous transactions accessing the same entries, and where speed is not essential. Even under these circumstances, applications that will run under distributed-ack scope require extra care in writing callbacks, whether for the cache or for transactions.

For transactions operating in regions of distributed- ack scope, when the VM originating the transaction leaves the distributed system, GemFire preserves the all-or-nothing consistency of the transaction. If the failure happens during the commit process while data is being distributed to other members of the distributed system, the recipients notice that the original VM has died. If any recipient got the complete transaction result, it forwards to the others. When the originating VM comes back up, for any mirrored regions, it acquires the transaction's result when it retrieves the regions' initial image. If no receiver got any committed data, all recipients drop the transaction.

Because transactions happen in the heap, if the VM originating the transaction goes down earlier, while the transaction is active, the transaction goes away and there is no memory of it. The transaction listener, for example, does not preserve any persistent data from ongoing transactions. The distributed system is still in a consistent state. The departure of any member of the distributed system, even one involved in an ongoing transaction, does not cause a write conflict.

Considerations for Mirrored Regions

Cache transactions on mirrored regions are supported. Depending on the application architecture and the frequency of updates, a mirror can increase the write conflicts for transactions writing to the mirrored region. Keys-values mirroring is more likely to cause conflicts than keys-only mirroring.

Considerations for Data and Application Design

To support high performance, GemFire transactions are optimistic; they are optimized for transactions that operate on their own data sets with little data sharing among transactions. If two transactions do modify the same keys, GemFire does not detect the conflict until commit time. Then the outcome depends on timing and the scope of the regions involved.

Using transactions effectively requires work during the data design phase, to structure your data in such a way that the transactions do not cross paths too often. Ideally, you can use optimistic transactions for most keys, to maximize performance, and do pessimistic locking on a few keys using GemFire's distributed lock service explicitly.

Cache Transactions and Application Plug-Ins

All of the standard GemFire application plug-ins work with transactions, and the transactions interface offers another plug-in that supports transactional operation. This section gives a detailed description of the transactional plug-in, TransactionListener, followed by some special considerations for using the standard GemFire plug-ins with transactions.

Transaction Listener

The GemFire cache transactions interface offers one plug-in, a specialized type of CacheListener named TransactionListener, to monitor transactional operations. A transaction listener is a synchronous event listener for transaction-related events invoked after the commit or rollback operation completes.

Only one transaction listener per cache is allowed, not one per region like the other cache listeners. That single transaction listener handles all the threads in all the regions of the cache. When the transaction ends, its thread calls the transaction listener to perform the appropriate follow-up for successful commits, failed commits, or voluntary rollbacks. The transaction that caused the listener to be called no longer exists at the time the listener code executes.

Transaction listeners have access to the transactional view and thus are not affected by non-transactional update operations. TransactionListener methods cannot make transactional changes or cause a rollback. They can, however, start a new transaction. Multiple transactions on the same cache can cause concurrent invocation of TransactionListener methods, so you need to implement methods that do the appropriate synchronizing of the multiple threads for thread-safe operation.

When a transaction listener is called, it reflects the cumulative action on each of the keys in the transaction. For example, if the transaction included three put operations to the same key and no other changes to that key, the listener would only see the last put operation, the one that actually affects the committed region. If a series of operations happened, the transactional listener would only see the final result, not the intermediate values.

A transaction listener can preserve the result of a transaction, perhaps to compare with other transactions, or for reference in case of a failed commit. When a commit fails and the transaction ends, the application cannot just retry the transaction, but must build up the data again. For most applications, the most efficient action is to just start a new transaction and go back through the application logic again.

The transaction listener can also be used to notify a cache writer when a transaction completes. A cache writer receives information about transactional operations, just as it does with other operations, but it does not know when the transaction completes without the help of a transaction listener.

The rollback and failed commit operations are local. When a successful commit writes to a distributed region, however, the transaction results are distributed to the remote VM. The transaction listener on the remote VM reflects the changes the transaction makes in the remote VM, not the local VM. Any exceptions thrown by the transaction listener are caught by GemFire and logged.

Behavior of Standard GemFire Application Plug-Ins With Transactions

The standard GemFire plug-ins operate as usual with transactions, but there are a few additional considerations.

CacheLoader

When a cache loader is called by a transaction operation, values loaded by the cache loader do not cause a conflict when the transaction commits.

CacheWriter

During a transaction, if a cache writer exists, each write operation (create, put, or destroy) in the transaction triggers a cache writer's related call (beforeCreate, beforeUpdate, beforeDestroy). Unlike the transaction listener, a cache writer receives each individual write operation. For example, if there are three put operations to the same key during a transaction, the cache writer would see all three, even though the cache would not change until the transaction commits. Regardless of the region's scope, a commit can invoke a cache writer only in the local cache, not in the remote caches. The netWrite operation is not invoked for transaction operations.

A cache writer cannot start a transaction, and it can only be notified when a GemFire transaction completes by using a transaction listener. If you want to roll back a transaction, for example, you need the cache writer to know about the rollback and behave appropriately. Your application could have the cache writer write to a buffer and, in case of a rollback or failed commit, dump the buffer. Alternatively, you could use JTA to handle it for you.

CacheWriter exceptions work as they do without transactions. The application receives the exception through the associated method invocation and makes its own decision about whether to abort the transaction.

CacheListener

On the VM that originated the transaction, the local cache listener's callbacks are triggered immediately as the region operations occur in the transaction; these callbacks happen before the commit. The EntryEvent that the callbacks receive has a unique transaction ID, so the cache listener can associate each event, as it occurs, with a particular transaction. For EntryEvents that are not part of a transaction, the transaction ID is null.

When GemFire runs under a global JTA transaction, the EntryEvents contain the GemFire transaction ID, not the JTA transaction ID.

If the region is distributed, in the remote VM the CacheListener's callbacks are triggered only after the transaction is committed. The EntryEvents received by the remote CacheListener include the transaction ID of the transaction in the originating VM.

JTA Transactions

JTA provides direct coordination between the GemFire cache and another transactional resource, such as a database. Using JTA, your application controls all transactions in the same standard way, whether the transactions act on the GemFire cache, a JDBC resource, or both together. By the time a JTA global transaction is done, the GemFire transaction and the database transaction are both complete.

Using GemFire in a JTA Transaction

Using GemFire with JTA transactions requires these general steps.

During configuration:

1. Configure the JDBC data sources in the cache.xml file.

2. Include the jar file for any data sources in your CLASSPATH.

At run-time:

3. Initialize the GemFire cache.

4. Get the JNDI context and look up the data sources.

5. Start a JTA transaction.

6. Execute operations in the cache and database as usual.

7. Commit the transaction.

The transactional view exists until the transaction completes. If the commit fails or the transaction is rolled back, the changes are dropped. If the commit succeeds, the changes recorded in the transactional view are merged into the cache. At any given time, the cache reflects the cumulative effect of all operations on the cache, including committed transactions and non-transactional operations.

JTA Transaction Limitations

The GemFire Enterprise Distributed implementation of JTA has these limitations:

- Only one JDBC database instance per transaction is allowed, although you can have multiple connections to that database.
- Multiple threads cannot participate in a transaction.
- Transaction recovery after a crash is not supported.

In addition, JTA transactions are subject to the limitations of GemFire cache transactions, which is discussed in the previous section. When a global transaction needs to access the GemFire cache, JTA silently starts a GemFire cache transaction.

GemFire Operation in Global Transactions

JTA syncs up multiple transactions under one umbrella by enabling transactional resources to enlist with a global transaction. The GemFire cache can register as a JTA resource through JTA synchronizations. This allows the JTA transaction manager to call methods like commit and rollback on the GemFire resource and manage the GemFire transactions. You can bind in JDBC resources so you can look them up in the JNDI context. When bound in, XAPooledDataSource resources will automatically enlist if called within the context of a transaction.

6 - Querying

GemFire Enterprise offers a standards-based querying implementation (OQL) for data held in the cache regions. Object Query Language, OQL, from ODMG (http://www.odmg.org) looks a lot like SQL when working with flat objects, but provides additional support for navigating object relationships, invoking object methods as part of query execution, etc. OQL queries can be executed on a single data region, across regions (inner-joined) or even arbitrary object collections supplied dynamically by the application. Query execution is highly optimized through the use of concurrent, memory-based data structures for both data and indexes. Applications that do batch updates can turn OFF synchronous index maintenance and allow the index to be optimally created in the background while the application proceeds to update the cache at memory speeds.

This chapter introduces the concepts of cache querying and presents examples of using the Object Query Language to retrieve cached data.

Querying Object Data

Object data is generally nested, and access to nested data requires drilling-down through the outer data layers. It follows that querying nested data also requires a drill-down approach. Data that is not “on the surface” must be brought into scope within a query in order to be accessed. This requires a special approach to querying that is subtly different from classic SQL querying. This section compares object data querying with querying classic relational data and provides a graphic view of the changes in data visibility as a GemFire cache query is run.

Querying Portfolios

The query package allows you to access cached regions and region data with SQL-style querying. If you have used SQL to query database tables, many of the concepts in this chapter will be familiar to you. There are differences, however, between the relational database table storage model and GemFire’s object storage model that translate into subtle but important differences in querying techniques. This section provides a comparison of the two storage models and their querying techniques. We use the portfolios example from the prior section. Each portfolio has an id, a status, a type, and map of associated positions. Each position represents a secId, a market value, and a quantity. In a database we would store this information in a portfolio table and a position table, connecting the two tables through primary and foreign keys. In a cache region, the position data would be stored as a map inside the portfolio data.

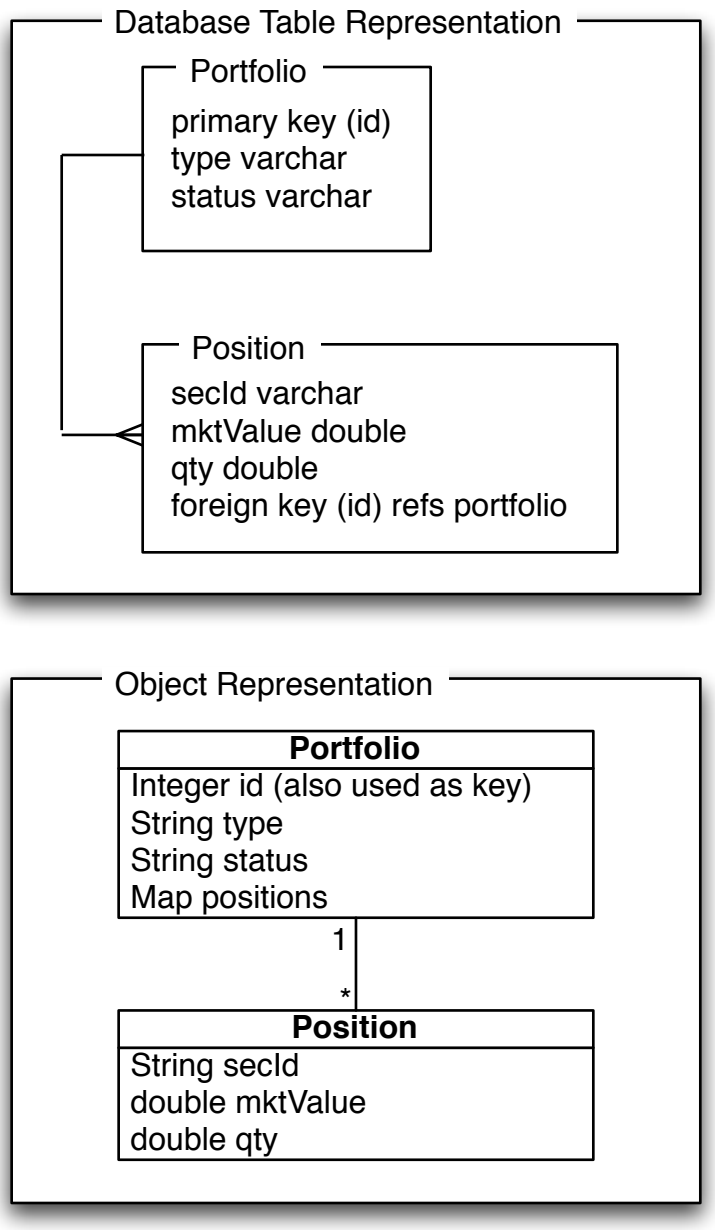


Figure: Contrasting database table and object representations

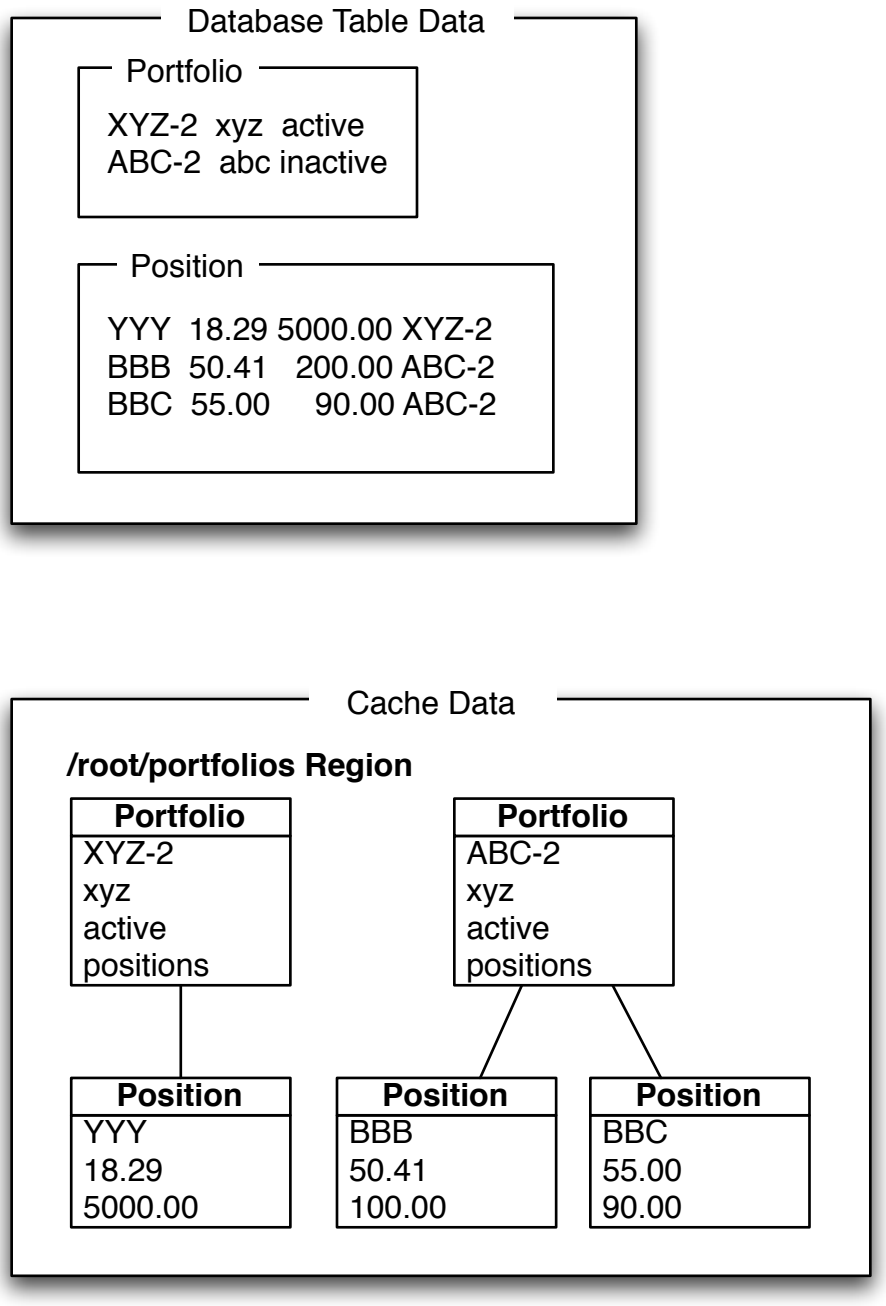


Figure: Contrasting database and cache data

As you can see, the classic database storage model is flatter, with each table standing on its own. We can access position data independent of the portfolio data or we can join the two tables if we need to by using multiple expressions in our FROM clause. With the object model, on the other hand, multiple expressions in the FROM clause are required for drilling down into the portfolio table to gain access to the position data. This can be helpful, as the position-to-portfolio relationship is implicit and does not require restating in our queries. It can also be a hindering factor when we want to ask general questions about the positions data, independent of the portfolio information. The two queries below illustrate these differences. We query the list of market values for all positions of active portfolios from the two data models with the statements below.

We query the list of market values for all positions of active portfolios from the two data models with the statements below.

SQL Query

```
SELECT mktValue
     FROM portfolio, position
     WHERE status='active'
     AND portfolio.id = position.id
```

Example: SQL query for list of market values for all positions of active portfolios

OQL Query

```
SELECT DISTINCT posnVal.mktValue
     FROM /root/portfolios, positions.values posnVal
     TYPE Position
     WHERE status='active'
```

Example: OQL query for list of market values for all positions of active portfolios

The difference between the queries reflects the difference in data storage models. The database query must explicitly join the portfolio and position data in the WHERE clause, by matching the position table's foreign key, id, with the portfolio table's primary key. In the object model, this one-to-many relationship was specified when we defined the positions Map as a field of the portfolio class.

This difference is reflected again when we query the full list of positions market values.

SQL Query

```
SELECT mktValue FROM position
```

Example: SQL query for the full list of market values

OQL Query

```
SELECT DISTINCT posnVal.mktValue
     FROM /root/portfolios, positions.values posnVal
     TYPE Position
```

Example: OQL query for the full list of market values

The position table is independent of the portfolio table, so our database query runs through the single table. The cached position data, however, is only accessible via the portfolio objects. The cache query aggregates data from each portfolio's individual positions Map to create the result set. Note: For the cache query engine, the positions data only becomes visible when the first expression in the FROM clause, /root/portfolios, is evaluated.

The next section explores this drill-down nature of cache querying more fully.

Data Accessibility in a GemFire Cache Query

Before any query is run, the only data in the cache that is directly accessible—in scope—are the regions and region methods. When a region or one of its attributes is referenced, the data returned by the reference comes into scope and can itself be referenced. For example, a region's keys and entries can be brought into scope by referencing /root/portfolios.keys and /root/portfolios.entries in the FROM clause. These references evaluate, respectively, to the collection of region keys and the collection of region Entry objects. The expression /root/portfolios evaluates by default to /root/portfolios.values, the collection of region entry values. This is used in the example query below.

This section shows the /root/portfolios data that is accessible as each part of a cache query is evaluated (the query used is the first one from the prior examples.) In each figure, the part of the query that has been evaluated is in bold. Our first figure shows the data that is in scope before any query is run. Note that region methods are visible.

Data in scope before query

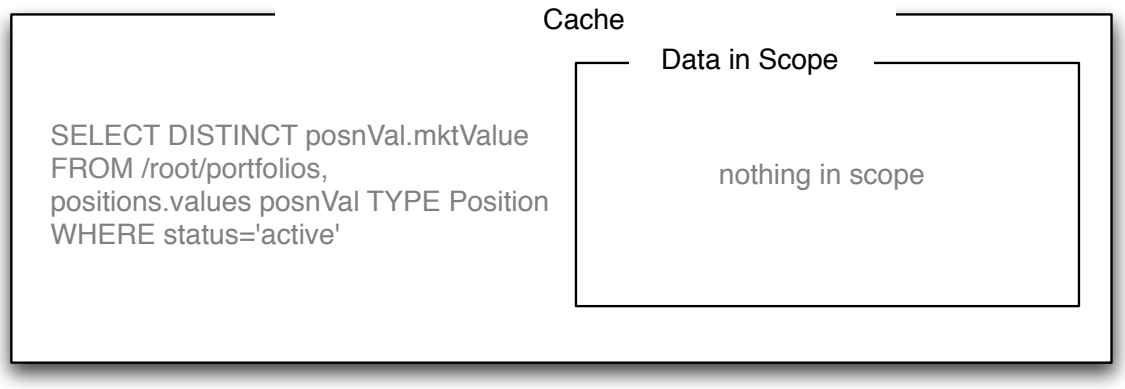


Figure: No scope is yet established by the query's FROM clause

Data in scope after first FROM expression is evaluated

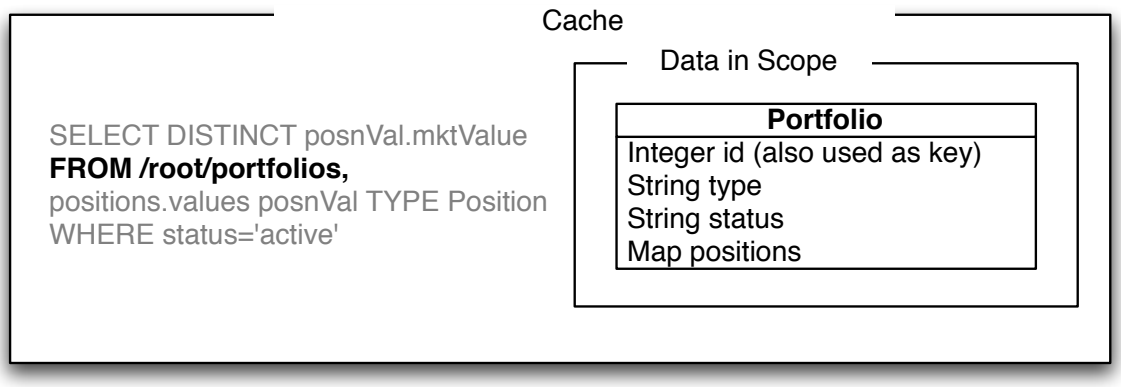


Figure: FROM clause establishes scope that includes all Portfolios

Data in scope after second FROM expression is evaluated

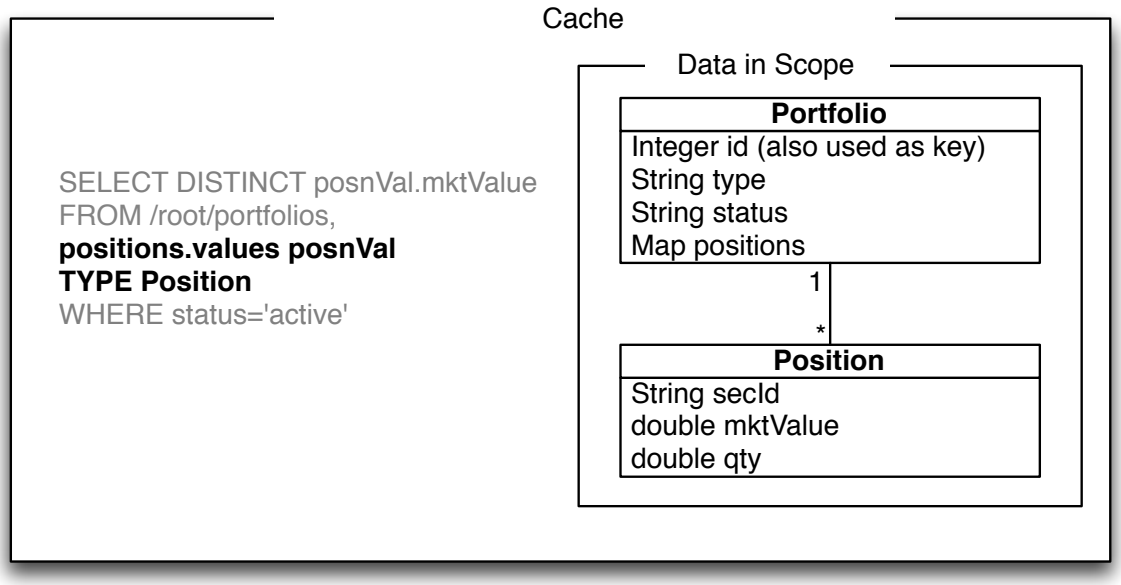


Figure: FROM clause establishes scope that includes all Portfolio positions

In cache querying, the FROM expression brings new data into the query scope.

Querying and the Cache

This section describes how the querying package interacts with the GemFire Enterprise Distributed cache.

Modifying Cache Contents

The query service is a data access tool. It does not provide any cache update functionality. To modify the cache based on information retrieved through querying, retrieve the entry keys and use them in the standard entry update methods.

Region Scope and MirrorType Recommendations

Querying and indexing only access the data that is available in the local cache. They do not perform netSearch, netLoad, or local cache loading operations. Depending on the region's Scope and MirrorType attribute settings, this could mean that your queries and indexes only see a part of the data that is defined for the region in the distributed cache. To ensure a complete data set for your queries and indexes, either set the region Scope to LOCAL or set the MirrorType to KEYS-VALUES.

A region with LOCAL Scope is completely contained in the local cache, so any access to the local region accesses the entire region data set. For a region with distributed scope, setting MirrorType to KEYS- VALUES pulls data from the rest of the distributed cache, so the local region contains the entire region data set. Any other mirroring configuration for a distributed region does not guarantee a complete local data set:

- With mirroring disabled, entries created in remote caches may not be available locally
- Mirroring of keys only guarantees that the region holds all of the distributed region's keys, but entry values can be missing

Depending on your use of the cache, the non-global distributed scopes may encounter race conditions during entry distribution that cause the local data set to be out of synch with the distributed region.

The Effects of scope and mirrorType on the data available for querying and indexing

This table summarizes the effects of region scope and mirroring settings on the data available to your querying and indexing operations.

MirrorType	none	keys	keys-values
Scope			
local	FULL DATA SET	N/A	N/A
distributed-no-ack	DATA SET MAY NOT BE COMPLETE	DATA SET MAY NOT BE COMPLETE	FULL DATA SET
distributed-ack	DATA SET MAY NOT BE COMPLETE	DATA SET MAY NOT BE COMPLETE	FULL DATA SET
global	DATA SET MAY NOT BE COMPLETE	DATA SET MAY NOT BE COMPLETE	FULL DATA SET

Table: The impact of region scope on query data

Accessing Cached Data

Accessing your cached data through the querying service is similar to how you would access database contents through SQL queries, but with some differences. Regions can contain nested data collections that are unavailable until referenced in the FROM clause, the query language in GemFire supports drilling down into nested object structures. This section discusses how to navigate to your cached data.

Basic Region Access

Querying and indexing only operate on local cache contents.. In the context of a query, the name of a region is specified by its full path starting with a forward slash ('/'), and delimited by the forward slash between regions names. Region names in a region path are restricted to alphanumeric characters or underscore characters only. From a region path, you can access the region's public fields and methods, referred to as the region's attributes. Thus, /root/portfolios.name returns "portfolios" and /root/portfolios.name.length returns 10. For information on how these attributes are resolved to the appropriate methods and fields, see attribute (page 210). You can access key and entry data through the region path. The expression, /root/portfolios.keys returns the Set of entry keys in the region, /root/portfolios.entries returns the Set of Region.Entry objects, and /root/portfolios.values returns the Collection of entry values. Any Collection returned from a SELECT statement is immutable, so methods to modify the Collection, such as add and remove, throw an UnsupportedOperationException when invoked.

Region Object Representation

In GemFire, when a region path evaluates to a Region in a query, the type of object it refers to has the following interface:

```
interface QRegion extends
                    com.gemstone.gemfire.cache.Region,
                    java.util.Collection
```

Example: Query regions can be treated as either a Collection or a Region

Thus, the interface inherits all the attributes and methods from both Region and Collection. When used as a Collection, the elements are the values stored in the region. This enables a query to use a region either as a collection of its values or as the actual region. The specific use in any instance depends on context. For example in this query, the /root/portfolios path is used as the Collection of portfolios in the region when we refer to positions.

```
SELECT DISTINCT positions FROM /root/portfolios
```

Example: Query region being treated as a Collection

The same path specification refers to the Region and therefore can be used to access attributes like keys that a Region understands, as in the following query.

```
SELECT DISTINCT * FROM /root/portfolios.keys
```

Example: Query region being treated as a Region

Drilling Down: 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 local cache and the attributes of those paths. New name spaces are brought into scope based on the FROM clause in the SELECT statement. In this query, for example, the FROM expression evaluates to the collection of entry values in /root/portfolios. This is added to the initial scope of the query and status is resolved within the new scope.

```
SELECT DISTINCT * FROM /root/portfolios
WHERE status='active'
```

Example: Bringing all Portfolios into scope

Each FROM clause expression resolves to a collection of objects available for iteration in the query expressions that follow. In the example above, /root/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.

```
SELECT DISTINCT ”type”
FROM /root/portfolios, positions.values posnVal
TYPE Position
WHERE posnVal.qty > 1000.00
```

Example: Bringing all portfolio Positions into scope

We could not change the order of the expressions in this FROM clause, the second expression depends on the scope created by the first expression.

Attribute Visibility

Within the current scope of a query, 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 locally cached regions and their attributes are in scope. This query is valid because keys resolves to the Region method, getKeys.

```
SELECT DISTINCT * FROM /root/portfolios.keys
```

Example: Referencing a region's keys

This query is valid because toArray resolves to the Collection method with the same name.

```
SELECT DISTINCT * FROM /root/portfolios.toArray
```

Example: Using a region as a Collection of values

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 con-

tain an attribute named positions, and the entry values collection (specified by /root/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 Collection*/

SELECT DISTINCT * FROM /root/portfolios.positions
/* INCORRECT */
```

Example: Attempting to use a Collection as a Portfolio

This following SELECT statement is valid because positions is an element of the entry value collection that is specified by /root/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 /root/portfolios
```

Example: Using an attribute of the entry bound by the FROM clause

You can also refer to positions inside the FROM clause after the /root/portfolios entry value collection is created. In this example, positions is an element of the /root/portfolios entry value collection and values is an attribute of positions:

```
SELECT DISTINCT posnVal
FROM /root/portfolios, positions.values posnVal
TYPE Position
WHERE posnVal.mktValue >= 25.00
```

Example: Using a variable bound in the FROM clause

After the comma in the FROM clause, /root/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. This is discussed further in The SELECT Statement (page 226).

7 - Data Availability and Fail-over

GemFire provides highly available data through the configuration of one or more 'mirror' (backup) caches. A 'mirror' is configured at a cache region level and synchronously receives all events on the region across the entire distributed system guaranteeing 100% backup of data at all times. It acts as a warm standby, so that when a failed application restarts and subscribes to a backed-up region, GemFire automatically loads all the data from the backup node.

The most common deployment model is one where the cache is collocated with the application in the same process. If the application process fails for any reason, the cache gets automatically disconnected from the distributed system. Upon application restart the cache reloads itself from the backup (lazily or at startup).

Making the data highly available through in-memory backups, though efficient, may not be sufficient in all situations. Some applications managing critical information in the cache may mandate that data be reliably managed on disk. GemFire accommodates such applications by optionally storing region entries persistently to the attached file system.

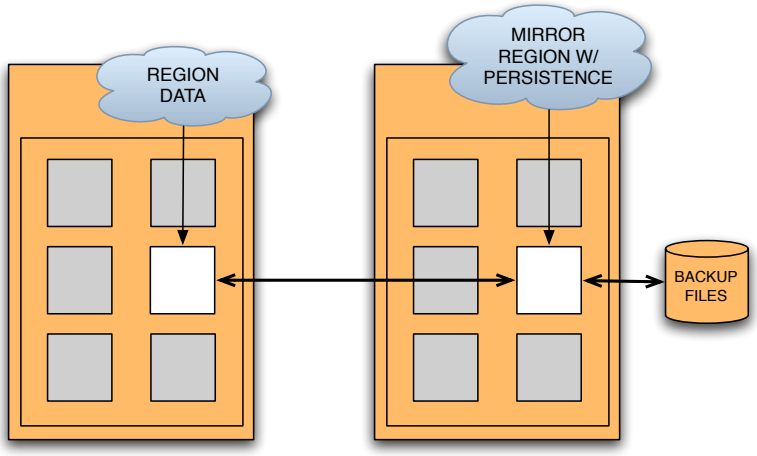


Figure: Using a persistent mirror for backup

Recovery, in general is extremely fast and done lazily. For instance, if an application with a disk region fails, the region is recovered immediately upon restart and the in-memory data cache builds up lazily. Similar is the case when a cache recovers from a 'mirror'. The recovery of a 'mirror', on the other hand, causes an 'initial image fetch' phase to be executed. This operation does a union of all data in the distributed system to build up the entire 'mirror' (backup). Applications connected to a 'mirror' can continue to access the cache without any impact.

Security

GemFire manages data in many nodes where access to data has to be protected. The security services provided by GemFire have the following characteristics:

On-the-wire protection (data integrity): This mechanism is used to prove that information has not been modified by a third party (some entity other than the source of the information). All communication between member caches can be made tamper proof again by configuring SSL (key signing). The use of SSL in GemFire communications is enabled in an all or nothing fashion.

Authentication: This refers to the protocol by which communicating entities prove to one another that they are acting on behalf of specific identities that are authorized for access. GemFire uses the J2SE JSSE (Java Secure Sockets Extension) provider for authentication. When SSL with mutual authentication is enabled, any application cache has to be authenticated by supplying the necessary credentials to the GemFire distributed system before it can join the distributed system. Authentication of connections can be enabled for each connection in a GemFire system. SSL can be configured for the locator service, the Console, the JMX agent and the GemFire XML server. The choice of providers for certificates, protocol and cipher suites are all configurable. The default use of the SUN JSSE provider can easily be switched to a different provider.

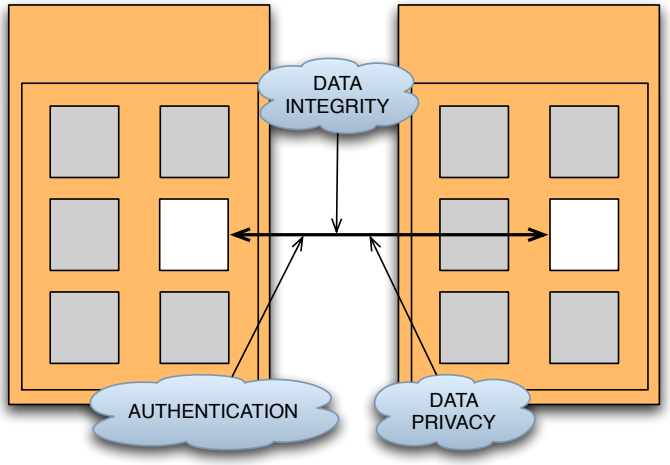


Figure: Using Using SSL for security

Confidentiality (data privacy): This feature ensures that information is made available only to users who are authorized to access it. All GemFire communication can be protected from eaves-droppers by configuring the SSL to use encryption (cipher suite). Applications can choose to use SSL for authentication, for encryption or to do both.

Role-based reliable data distribution

GemFire provides a novel, declarative (user-defined) approach for managing data distribution with the required levels of reliability and consistency across several hundreds or even thousands of nodes. Application architects can define 'roles' relating to specific functions and identify certain roles as 'required roles' for a given operation/application. GemFire instances are assigned roles. Applications connected to a GemFire instance identify which roles are required to be present in the distributed system for them to operate normally. If for some reason none of the instances that have the required roles are available then the application can be configured to respond in one of the following ways

- block any cache operations
- allow certain specific cache operations
- allow all cache operations
- disconnect and reconnect for a specified number of times to check if the required roles are back online

For instance, 'DB Writer' can be defined as a role that describes a member in the GemFire distributed system that writes cache updates to a database. The 'DB Writer' role can now be associated as a 'required role' for another application (Data feeder), whose function is to receive data streams (for e.g., price quotes) from multiple sources and pass on to a database writer. Once the system is configured in such a fashion, the data feeder will check to see if at least one of the applications with role 'DB Writer' is online and functional before it propagates any data updates. If for some reason, none of the 'DB Writers' are available, the price feeder application can be configured to respond as described above.

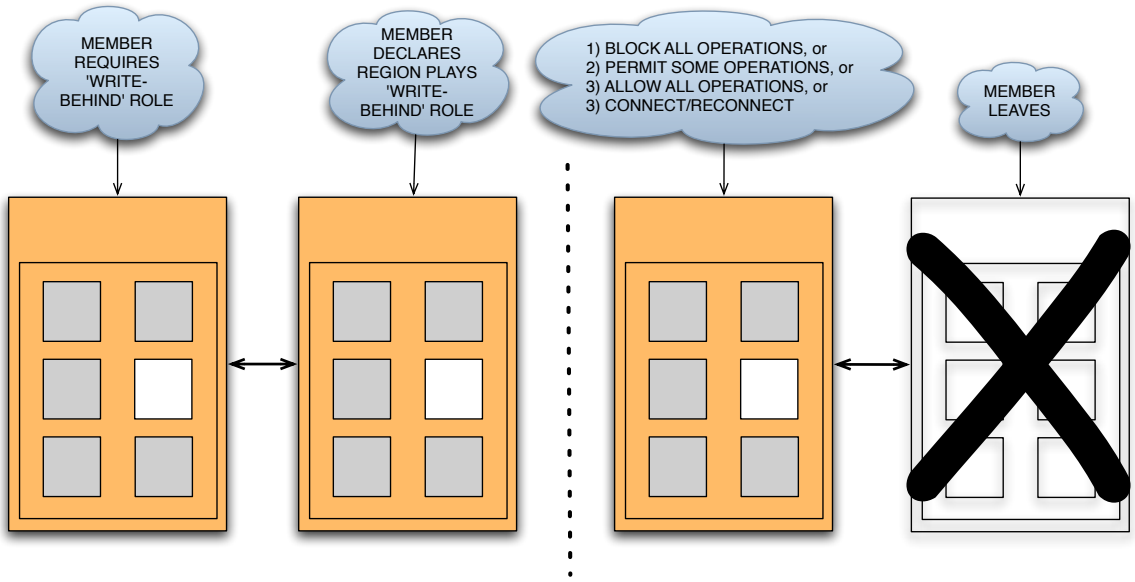


Figure: Responding to a required role exiting the system

The role declarations can also be used to enable a GemFire system to automatically handle and recover from issues such as network segmentations, which cause a distributed system to become disjointed into two or more partitions. In such a scenario, each member in a disjointed partition evaluates the availability of all the required roles in that partition, and if all such roles are available, then that partition automatically reconfigures itself and sustains itself as an independent GemFire distributed system. On the other hand, if all the required roles are not found in a partition, then the primary member in that partition can be configured to disconnect and reconnect to the GemFire distributed system a specified number of times. This is of

course is done with the expectation that network partition would be addressed within a short period of time and all required roles would become available again. If this reconnect protocol fails, the member shuts down after logging an appropriate error message. With this 'self-healing' approach, a network segmentation/partitioning is handled by the distributed without any human intervention.

The operational reliability and consistency of the system can be managed without resorting to overly pessimistic 'all or nothing' style policies (supported by other distributed caching in the market) that have no application specific context. Traditional distributed caching solutions do not provide an architect with the ability to define critical members in a distributed system and cannot guarantee that critical members are always available prior to propagating key data updates leading to missed messages and data inconsistencies. The GemFire role-based model offers the perfect balance of consistency, reliability and performance, without compromise on any of these dimensions.

Slow or unresponsive receivers/consumers

In most distributed environments, overall system performance and throughput can be adversely impacted if one of the applications/receivers consumes messages at a rate slower than that of other receivers. For instance, this may be the case when one of the consumers if it is not able to handle a burst of messages, due to its CPU intensive processing on a message-by-message basis. With GemFire Enterprise, a distribution timeout can be associated with each consumer, so that if a producer does not receive message acknowledgments within the timeout period from the consumer, it can switch from the default synchronous communication mode to an asynchronous mode for that consumer.

When the asynchronous communication mode is used, a producer batches messages to be sent to a consumer via a queue, the size of which is controlled either via queue timeout policy or a queue max size parameter. Events being sent to this queue can also be conflated if the receiver is interested only in the most recent value of a data entity.

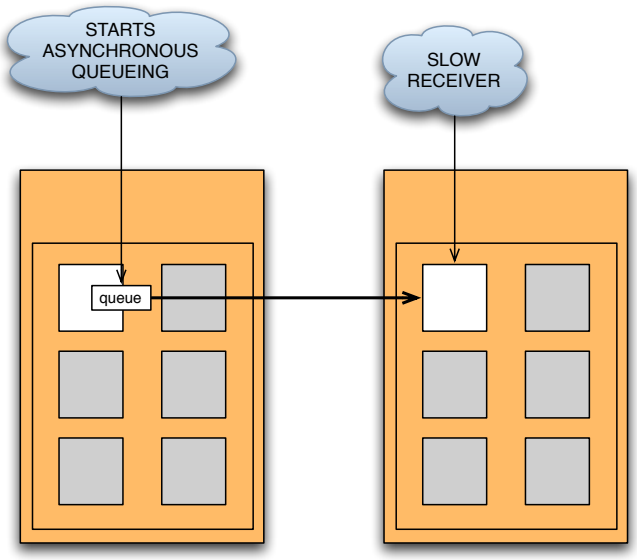


Figure: Slow receiver causes sender to start asynchronous queuing

Once the queue is empty, the producer switches back to the synchronous distribution mode, so that message latencies are removed and cache consistency is ensured at all times.

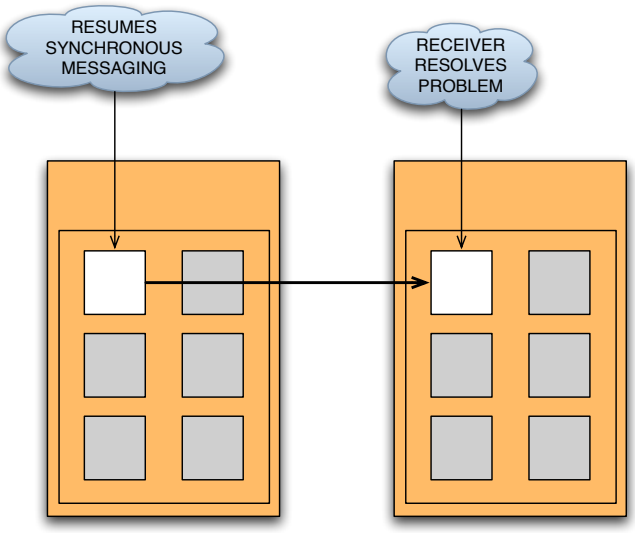


Figure: Slow receiver catches up, sender resumes synchronous messaging

On the other hand, if either the queue timeout or the queue max size condition is violated, the producer sends a high priority message (on a separate TCP connection) asking the consumer to disconnect and re-connect afresh into the GemFire system, preferably after resolving the issues that caused the consumer to operate slowly.

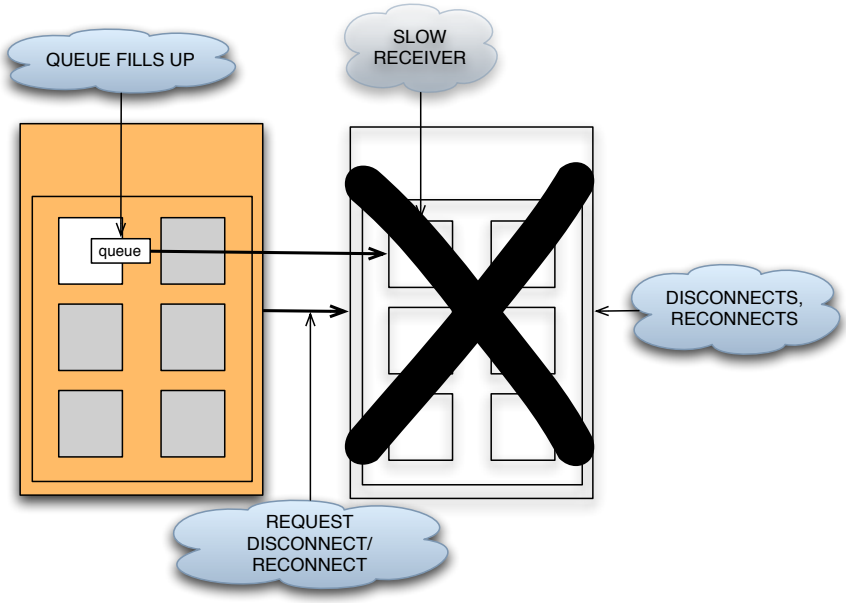


Figure: Sender queue fills up, sender requests slow receiver to exit and reconnect

If in an extreme situation, the consumer is not able to receive even the high priority messages, the producer logs warning messages, based on which a system administrator can manually fix the offending consumer. If it is not fixed manually, the GemFire system will eventually remove the consumer from the distributed system based on repeated messages logged by a producer. In this fashion, the overall quality of service across the distributed system is maintained by quarantining an ailing member.

8 - GemFire Administration

GemFire provides a membership API that can be used to detect a split-brain scenario. The solution to the problem requires three members at a minimum. When a member detects ungraceful exit of another member, it announces this to other members. Similarly, the other members do the same. If more than one member indicates the unexpected exit of the same member, the member is presumed unreachable. On the other hand, the member who lost its network link, will notice that all the other members have ungracefully left the system and can take a different course of action.

System Management Tools

GemFire facilitates managing and monitoring a distributed cache system and its member caches through three methods:

- The GemFire Console
- JMX APIs and Agent
- The GemFire command line tool

GemFire Console (gfc)

The GemFire Console is a tool for administering, inspecting, monitoring and analyzing distributed GemFire systems, GemFire managers and applications.

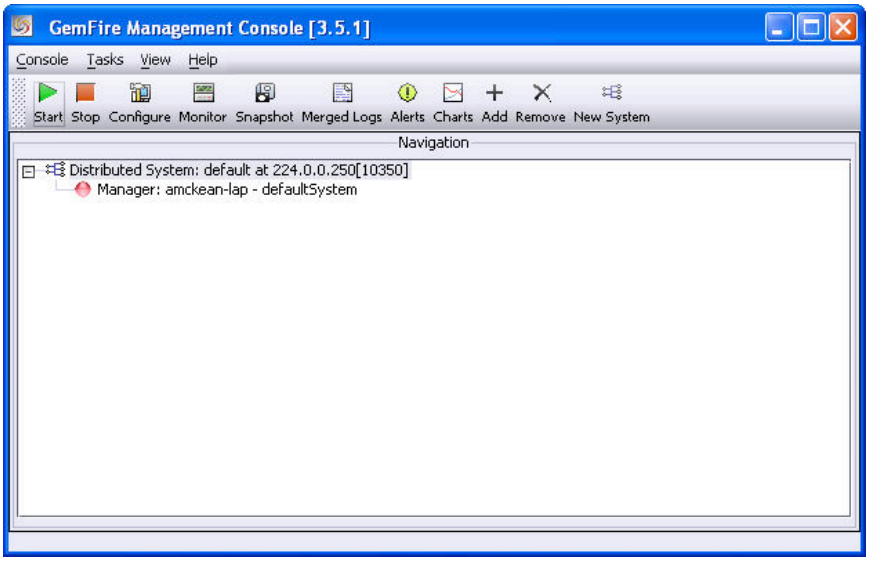


Figure: The GemFire Console is a general-purpose system monitor

A single instance of the GemFire Console monitors all members of a distributed GemFire system, providing the ability to view the following:

- Configuration of the entire distributed cache system
- Configuration and runtime settings for each member cache
- The contents of application caches
- Data and system statistics garnered from all members of a distributed system

The Console can be used by administrators and developers to launch remote GemFire systems and to modify runtime configurations on remote system members.

JMX

The Java Management extensions (the JMX specification) define the architecture, design patterns, APIs, the services for application / network management and monitoring for Java based systems. GemFire provides access to all its cache management and monitoring services through a set of JMX APIs. JMX enables GemFire to be integrated with Enterprise network management systems such as Tivoli, HP Openview, Unicenter, etc. Use of JMX also enables GemFire to be a managed component within an application server that hosts the cache member instance.

GemFire exposes all its administration and monitoring APIs through a set of JMX MBeans. An optional JMX agent process can be started to manage the entire distributed system from a single entry point. The agent provides HTTP and RMI access for remote management consoles or applications, but also makes it possible to plug-in third party JMX protocol adapters, such as SNMP. The JMX health monitoring APIs allows administrators to configure how frequently the health of the various GemFire components such as the distributed system, system manager processes and member caches. For instance, the distributed system health is considered poor if distribution operations take too long, cache hit ratio is consistently low or the cache event processing queues are too large. The JMX administrative APIs can be used to start, stop and access the configuration of the distribution system and its member caches right to the level of each cached region. The JMX runtime statistics API can be used to monitor statistics gathered by each member cache. These statistics can be correlated to measure the performance of the cache, the data distribution in the cache and overall cache scalability.

GemFire command line utility

The GemFire command-line utility allows you to start, stop and otherwise manage a GemFire system from an operating system command prompt. The gemfire utility provides an alternative to use of the GemFire Console (gfc) and allows you to perform basic administration tasks from a script. However, all GemFire administrative operations must be executed on the same machine as the GemFire system and only apply to a single GemFire system member.

Statistics

GemFire has a very flexible and revealing statistics representation. During execution, a distributed system has the ability to record many types of data regarding the performance and behavior of the system:

- Cache operations - information regarding the type and number of cache operations and how much time they consume
- Messaging - statistics regarding message traffic between the VM and other distributed system members
- Virtual machine - statistics describing the VM's memory usage. These can be used to find Java object leaks
- Statistics - statistics that show how much time is spent collecting statistics
- Operating system process - operating system statistics on the VM's process. Can be used to determine the VM's cpu, memory, and disk usage
- Member machine - operating system statistics on the VM's machine: total cpu, memory, and disk usage on the machine

In addition to the predefined statistics, application developers can define custom, application-specific statistics. Any type of information can be recorded at runtime with these custom statistics and they are treated by

the tools (see VSD below) as first-class statistics that can be displayed and analyzed alongside the other, predefined statistics.

Visual Statistics Display

The Visual Statistics Display (VSD) is a graphical user interface for the viewing the statistics that GemFire records during application execution. It can be used to chart any GemFire statistics over time, overlay different statistics onto the same chart, and hone in on specific aspects of system performance by filtering and compressing statistics.

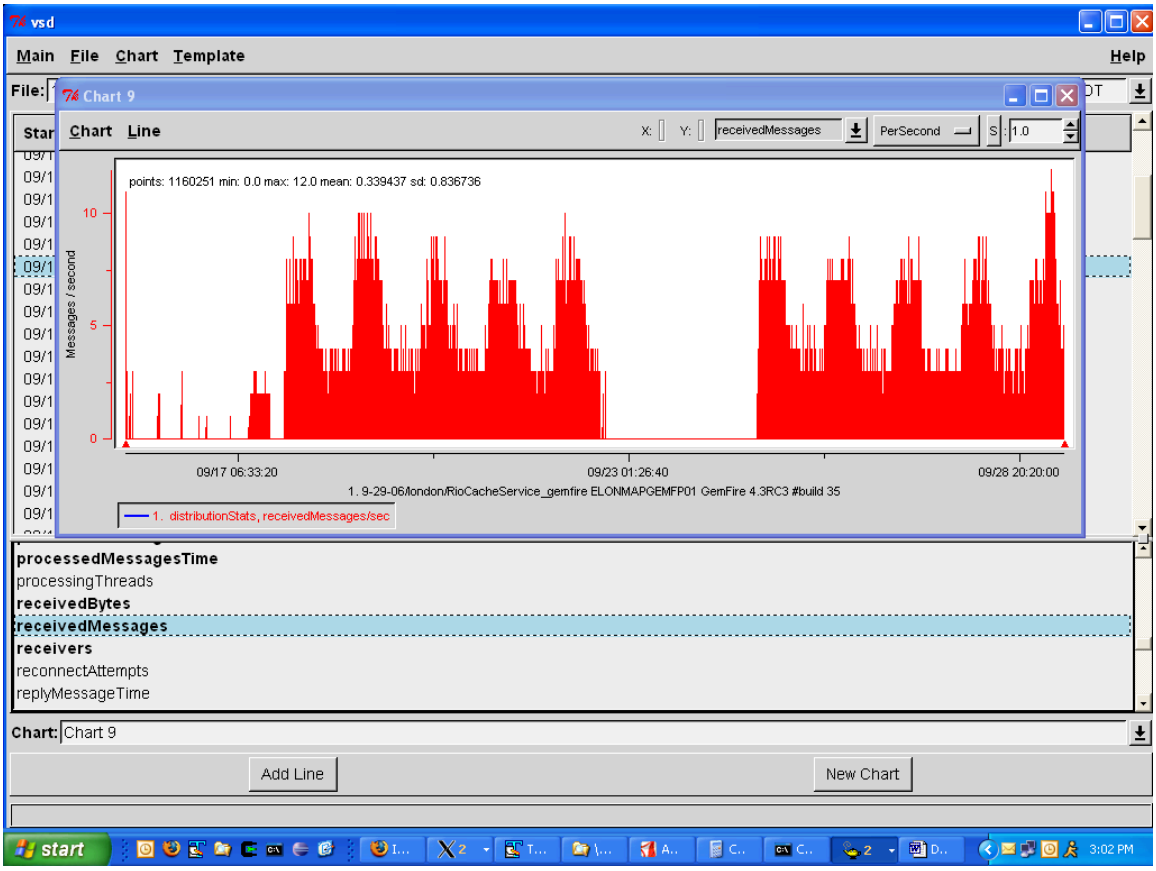


Figure: Visual Statistics Display is used for charting and analyzing statistics

VSD allows the statistics files to be merged into one view. This allows the developer/administrator to analyze dependent behaviors among different distributed system members.

GemFire Admin and Health API

The GemFire Administration API for Java provides programmatic access to much of the same functionality provided by the GemFire Console for a single distributed system. This section gives an

overview of the primary interfaces and classes that are provided by the API package. The administration API allows you to configure, start, and stop a distributed system and many of its components. The API is made up of distributed system administration, component administration, and cache administration. In addition to the core components listed here, the administration API provides interfaces to issue and handle system member alerts and to monitor statistics.

The health monitoring API allows you to configure and monitor system health indicators for GemFire Enterprise Distributed Systems and their components. There are three levels of health: good health that indicates that all GemFire components are behaving reasonably, okay health that indicates that one or more GemFire components is slightly unhealthy and may need some attention, and poor health that indicates that a GemFire component is unhealthy and needs immediate attention.

Because each GemFire application has its own definition of what it means to be “healthy”, the metrics that are used to determine health are configurable. It provides methods for configuring the health of the distributed system, system managers, and members that host Cache instances. Health can be configured on both a global and per-machine basis. It also allows you to configure how often GemFire’s health is evaluated.

The health administration APIs allow you to configure performance thresholds for each component type in the distributed system (including the distributed system itself). These threshold settings are compared to system statistics to obtain a report on each component’s health. A component is considered to be in good health if all of the user-specified criteria for that component are satisfied. The other possible health settings, okay and poor, are assigned to a component as fewer of the health criteria are met.

Logging

GemFire's logging facility is based on the Apache Log4j project. It is an extensible logging library for Java. With Log4j it is possible to enable logging at runtime without modifying the application binary. Log4j lets the developer control which log statements are output with arbitrary granularity. The Log4j package is designed so that these statements can remain in shipped code without incurring a heavy performance cost. Logging behavior can be controlled by editing a configuration file, without touching the application binary.

Java VM performance

There are several JVM parameters that will affect performance. Using VSD to identify bottlenecks and monitor the runtime behavior may suggest tweaking the JVM's configuration with one of the following parameters:

- VMHeapSize - sets both the maximum and initial memory sizes or a Java application
- MaxDirectMemorySize - limits the amount of memory the VM allocates for the NIO
- direct buffers
- UseConcMarkSweepGC and UseParNewGC - controls parallel garbage collection
- DisableExplicitGC - Disables periodic gc performed by VM

Appendix A - Code Examples

Configuration

Distributed system properties can be set programmatically (using the GemFire API), or can be declared in the gemfire.properties file.

```
# in gemfire.properties file
name=default
cache-xml-file=cache.xml
system-directory=defaultSystem
mcast-address=224.0.0.250
mcast-port=10333
mcast-ttl=32
locators=
license-file=gemfireLicense.zip
license-type=evaluation
ssl-enabled=false
ssl-ciphers=any
ssl-protocols=any
ssl-require-authentication=true
async-max-queue-size=8
async-queue-timeout=60000
async-distribution-timeout=0
async-allow-conflation=true
socket-buffer-size=32768
socket-lease-time=15000
ack-wait-threshold=15
conserve-sockets=true
log-level=config
log-file-size-limit=0
log-file=system.log
log-disk-space-limit=0
statistic-sample-rate=1000
statistic-sampling-enabled=true
statistic-archive-file=statArchive.gfs
archive-file-size-limit=0
archive-disk-space-limit=0
```

Example: Configuring a distributed system with the gemfire.properties file

Individual member configuration parameters can be set programmatically (using the GemFire API), or can be declared in the member's cache.xml file.

```
<!-- in cache.xml file -->
<cache>
  <vm-root-region name='items'>
    <region-attributes scope='distributed-ack'>
    </region-attributes>
  </vm-root-region>
  <vm-root-region name='offerings'>
    <region-attributes mirror-type='keys-values'>
    </region-attributes>
  </vm-root-region>
</cache>
```

Example: Configuring a member with the cache.xml file

A Wrapper for the GemFire API

The following 'wrapper' code encapsulates many of the key elements of the GemFire Java API. It shows how to connect to a distributed system, how to create and configure a cache and a region in the cache, and it demonstrates many of the operations that an application can perform on cache data.

Connecting to a distributed system and creating a cache

```
// GemfireWrapper.java
public class GemFireWrapper {
  protected static GemFireWrapper instance;
  protected DistributedSystem system;
  protected Cache cache;

  protected GemFireWrapper() {
    Properties properties = new Properties();
    properties.setProperty("mcast-address", "224.0.0.250");
    properties.setProperty("mcast-port", "10350");
    system = DistributedSystem.connect(properties);
    try {
      cache = CacheFactory.create(system);
    }
    catch(CacheExistsException e) { System.out.println(e); }
    catch(RegionExistsException e) { System.out.println(e); }
    catch(GatewayException e) { System.out.println(e); }
    catch(CapacityControllerException e) { System.out.println(e); }
    catch(CacheWriterException e) { System.out.println(e); }
    catch(TimeoutException e) { System.out.println(e); }
  }

  public static GemFireWrapper getInstance() {
    if(instance == null) {
      instance = new GemFireWrapper();
    }
    return instance;
  }
```

Example: Using a utility 'wrapper' class to encapsulate GemFire API calls

Creating a region

```
// GemfireWrapper.java: region creation

  public boolean createRegion(String name,
    Scope scope, MirrorType mirrorType, boolean persists, File[] dirs,
    boolean statsEnabled, LRUAlgorithm controller)
  {
    AttributesFactory factory = new AttributesFactory();
    factory.setScope(scope);
    factory.setMirrorType(mirrorType);
    if(persists && (dirs != null)) {
      factory.setPersistBackup(persists);
      factory.setDiskDirs(dirs);
    }
    if(dirs != null) {
      factory.setDiskDirs(dirs);
    }
    factory.setStatisticsEnabled(statsEnabled);
    if(controller != null) {
      factory.setCapacityController(controller);
    }
    RegionAttributes ra = factory.createRegionAttributes();
    try {
      cache.createVMRegion(name, ra);
      return true;
    }
    catch(RegionExistsException e) { return false; }
    catch(TimeoutException e) { return false; }
  }
```

Example: Creating a region with a parameterized 'wrapper' method

Region data operations

```
// GemfireWrapper.java: region operations

  public boolean createObject(String s, Object k, Object v) {
    Region region = cache.getRegion(name);
    Region region = r; Object key = k; Object value = v;
    try {
      region.create(key, value);
      return true;
    }
    catch(EntryExistsException e) { return false; }
    catch(CacheWriterException e) { return false; }
    catch(CapacityControllerException e) { return false; }
    catch(TimeoutException e) { return false; }
  }

 public boolean updateObject(String name, Object key, Object value) {
    Region region = null;
    region = cache.getRegion(name);
    try {
      region.put(key, value);
      return true;
    }
    catch (CacheWriterException e) { return false; }
    catch (CapacityControllerException e) { return false; }
    catch (TimeoutException e) { return false; }
    return false;
  }

  public Object getObject(String name, Object key) {
    Region region = cache.getRegion(name);
      try {
        return region.get(key);
      }
      catch(CacheLoaderException e) { return null; }
      catch(TimeoutException e) { return null; }
  }

  public boolean destroyObject(String name, Object key) {
    Region region = cache.getRegion(name);
    try {
      region.destroy(key);
      return true;
    }
    catch(EntryNotFoundException e) { return false; }
    catch(CacheWriterException e) { return false; }
    catch(TimeoutException e) { return false; }
```

Example: Using the GemFire API Region operations

Installing plugins

```
// GemfireWrapper.java: Install plugins and configure eviction

  public void setListener(String name, CacheListener listener) {
    Region region = cache.getRegion(name);
    AttributesMutator mutator = region.getAttributesMutator();
    mutator.setCacheListener(listener);
  }

  public void setEntryIdleTimeout(String name,ExpirationAttributes a){
    Region region = cache.getRegion(name);
    AttributesMutator mutator = region.getAttributesMutator();
    mutator.setEntryIdleTimeout(expAttrs);
  }

  public void setWriter(String name, CacheWriter writer) {
    Region region = cache.getRegion(name);
    AttributesMutator mutator = region.getAttributesMutator();
    mutator.setCacheWriter(writer);
  }

  public void setLoader(String name, CacheLoader loader) {
    Region region = cache.getRegion(name);
    AttributesMutator mutator = region.getAttributesMutator();
    mutator.setCacheLoader(loader);
  }

  public void close() {
    cache.close();
  }
}
```

Example: Using the GemFire API to install plugins at runtime

Using a transaction

```
// Transaction programming
GemFireWrapper wrapper = GemfireWrapper.getInstance();
CacheTransactionManager manager = cache.getCacheTransactionManager();

manager.begin();
  Item item = wrapper.getObject("items", "car");
  Item txnItem = CopyHelper.copy(item);
  txnItem.setDescription("limo");
try {
  manager.commit();
}
catch (CommitConflictException ex) {
}
```

Example: Beginning a transaction, operating on data, committing the transaction

Taken out:

Performance-oriented applications such as real-time trading systems benefit from in-memory database technologies for managing unified orderbooks, messaging for capturing real-time data, and for querying and receiving updates to streaming data. Online web applications that have hundreds of thousands of users can benefit from load balancing inherent in a distributed cache and from the query services driven by database semantics. Client applications that need to run offline can take advantage of a persistent local cache to automatically synchronize with a server when it reconnects to the server.