




## GemFire Enterprise

This document describes the concepts, architecture and implementation topologies of GemFire Enterprise, a core component of the GemFire Enterprise Data Fabric (EDF). This is targeted at technical architects and developers evaluating or implementing the GemFire Enterprise. It addresses some important aspects that typically impede an architect such as architectural resilience, systems management, robustness, and security.



## Table of Contents

1. OVERVIEW .....	3
2. INTRODUCTION TO GEMFIRE ENTERPRISE .....	3
3. ARCHITECTURAL OVERVIEW AND CONCEPTS .....	5
3.1 DISTRIBUTED CACHING CONCEPTS .....	5
3.2 CACHE TOPOLOGIES AND STORAGE MODEL .....	7
3.3 QUERYING .....	11
3.4 TRANSACTIONS .....	11
4. DATA DISTRIBUTION AND CONSISTENCY MODELS .....	12
5. HIGH AVAILABILITY AND FAIL-OVER .....	14
6. HETEROGENEOUS DATA MANAGEMENT .....	15
6.1 C++, C#, AND .NET SUPPORT AND INTEROPERABILITY .....	15
7. ENTERPRISE CONNECTIVITY .....	16
7.1 CONNECTING WITH DATA SOURCES .....	16
7.2 DISTRIBUTED EVENT NOTIFICATION SERVICES .....	16
8. SYSTEM MANAGEMENT .....	17
8.1 TROUBLESHOOTING TRACING AND TUNING .....	17
8.2 SYSTEM MANAGEMENT TOOLS .....	18
9. SECURITY FRAMEWORK .....	19
10. USE-CASES .....	20
10.1 GEMFIRE IN A PROGRAM TRADING ENVIRONMENT .....	20
10.2: GEMFIRE IN AN ONLINE FIXED INCOME SECURITIES TRADING J2EE PORTAL ...	22
10.3 GEMFIRE DATA GRID FOR RISK ANALYTICS .....	25



## 1. OVERVIEW

Data management looms as a major challenge across most IT organizations. Some of the key challenges include:

- managing constantly growing volumes of data
- increased need to manage data across distributed commodity hardware and cheap memory
- increased performance and low-latency requirements
- more stringent service level agreements
- multiple platforms, languages (Java, C++, .NET) and data types (relational, streaming, objects)
- information delivery to clients across different geographies and networks
- optimizing network bandwidth and data distribution
- scaling to sporadic loads

Given these challenges and the advent of commodity distributed hardware like blades, an in-memory data fabric such as GemFire Enterprise is an ideal solution to address performance and scalability challenges encountered by IT organizations today. A data fabric acts as operational middleware for data. It provides an architectural layer that stores data from databases and backend systems in memory, ensuring that these systems are not in the critical path of distributed low-latency operations. Constantly reducing memory prices and the mushroom-growth of distributed applications make RAM the preferred platform of critical data operations, relegating the disk to archival and warehousing functions. By adopting such memory-based technologies, SOA and grid environments can benefit from 'high-performance data virtualization' - a popular oxymoron until now. In other words, applications can transparently deal with extremely large volumes of disparate data and achieve this transparency with almost zero-latency.

## 2. INTRODUCTION TO GEMFIRE ENTERPRISE

GemFire Enterprise is an ultra high-performance, distributed data management platform that harnesses resources across a network to form a real-time data fabric or data grid. By managing data in memory, GemFire enables extremely high-speed data sharing, turning a network of machines into a single logical unit. Following is a brief description of the functional aspects of GemFire Enterprise:

**High Performance Data Caching:** A cache is a temporary place to manage data to boost data access performance. Most modern day applications or middle-tier products implement some form of caching. GemFire provides a more valuable cache that can scale beyond the limits of process heap, is highly available, is distributed and synchronizes the data across several nodes and backend applications.

**Data distribution and notification:** This feature refers to the ability to manage the data in a distributed fashion often partitioned across many nodes. Distributed caching solutions are often expected to synchronize and manage the consistency of data across several nodes. GemFire has

the ability to control the memory consumption of specific cache instances, and to optimize the instance so that it will contain only the most relevant information at any given time.

As data in the cache gets updated, notifications can be sent to the applications that have explicitly registered interest in these updates across many nodes.

**Data virtualization:** One of the key concepts in areas like grid computing is virtualization; the ability to make distributed resources appear as one and to allow resources to plug into and out of the Grid as needed. "Data Virtualization" is the ability to make many different types of data sources appear as if it were just one. It abstracts the developer from data source heterogeneity, data formats and location of data. GemFire, through a plug-in framework, abstracts applications from having to deal with data sources directly. By distributing the data across many nodes based on demand, GemFire provides location transparency.

**High availability:** Real-time applications such as market data systems in financial services can use GemFire as the repository for market data or trade information to achieve the levels of high performance required. In such application usage scenarios, data availability has to be guaranteed with minimal overhead. GemFire provides various schemes for high availability, from n-way replication to use of persistent storage. GemFire provides features to ensure there is no single point of failure (SPOF) in the system.

**Real-time Data Integration:** While most integration products provide tools to model interactions between applications, seldom can they integrate data intensive applications without compromising performance and consistency of data. GemFire, through its distributed data fabric connects Java, C++, and .NET applications very efficiently by placing the right data on the node where it is most often used. The use of neutral data representations for objects reduces the pain associated with most transformations, e.g., O-R mapping.

The GemFire Enterprise high-level architecture is shown in figure 1. The architectural concepts are explored in detail in the following sections.

## Architecture

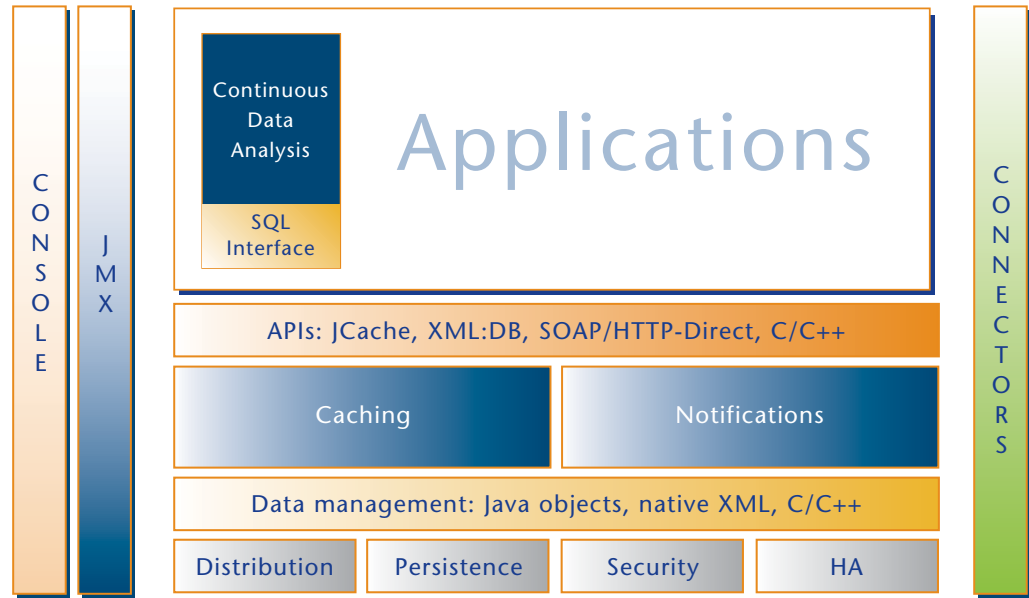


Figure 1: GemFire high-level architecture

### 3. ARCHITECTURAL OVERVIEW AND CONCEPTS

#### 3.1 Distributed Caching Concepts

Typically, a GemFire distributed system consists of any number of member caches that are connected to one another in a peer-to-peer fashion, such that each member cache is aware of the availability of every other member at any time. The GemFire distributed cache API presents the entire distributed system as if it were just one logical cache (see figure 2), =abstracting the location

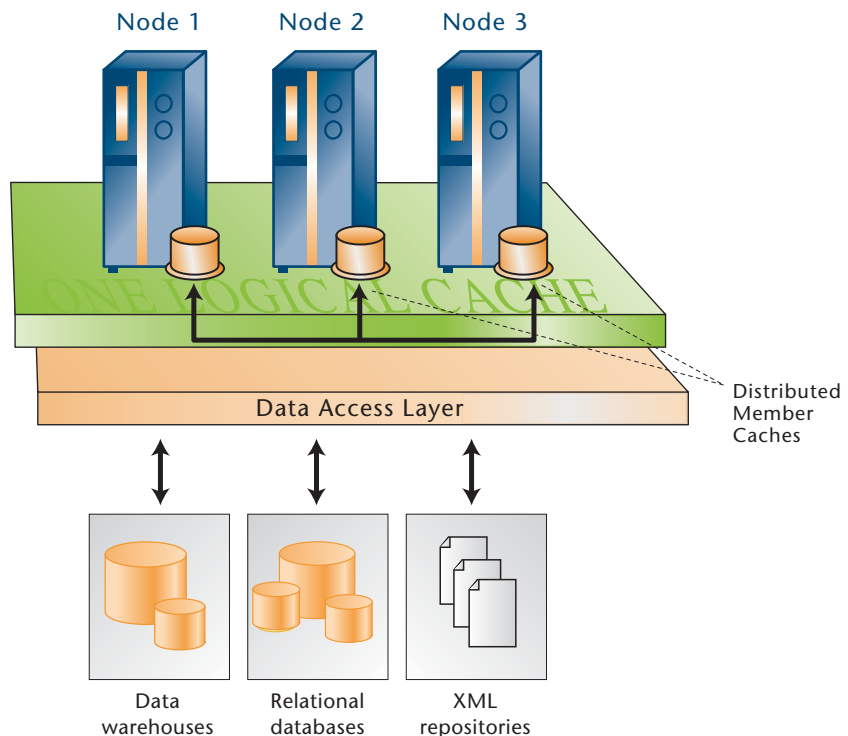


Figure 2: Logical view

of the data or the data source from the developer. Each distributed cache system is identified by an IP address and port.

A member cache uses this address and port specification to discover all the members connected to it. By default, GemFire uses IP multicast for the discovery process, while all member to member communication is based on TCP. For deployment environments where the use of multicast may not be an option, due to internal network policies or the requirement to span multiple subnets, GemFire provides an alternative approach through a framework of "locators".

A "locator" is a very light-weight GemFire service that keeps track of all member connections (figure 3). It is aware of any member joining or leaving the system at any time. Member caches connect to the distributed system by way of a locator when multicast is not being used. Any number of "locators" can be started for fault-tolerance.

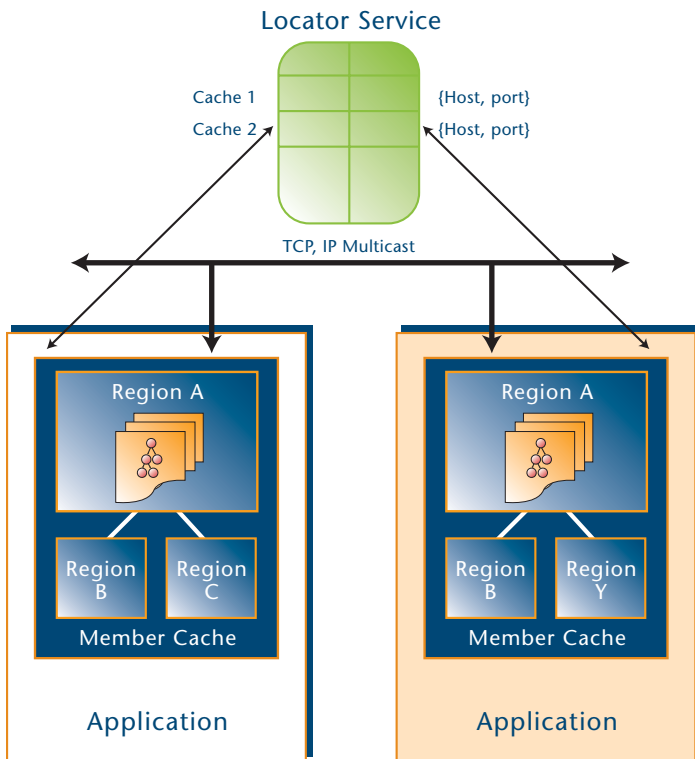


Figure 3: GemFire locator service

Each member cache connection to the distributed system has a unique name, which allows administrators to more easily monitor each cache. Each member cache is made up of one or more cache "Regions", which may further contain sub-regions, effectively creating hierarchical regions. A cache region extends the `java.util.Map` interface and manages a collection of application objects. Each region carries a set of configuration attributes that control where (the physical location) the data is managed, the distribution characteristics and the eviction and consistency models.

Data communication between member caches is intelligent. Each member cache has enough topological information to know which members share the regions it has defined. The distribution layer thus intelligently routes the messages only to the right nodes.

The distribution system is designed such that members can join and leave at any time without impacting other member caches. For instance, a heavily loaded clustered "stateful" application can be easily scaled by adding more nodes dynamically. Similarly, a reduction in the application load would allow administrators to remove member caches.

### **3.2 Cache topologies and storage model**

Given the range of possible problems that a middle-tier cache needs to address, a product that provides architectural flexibility is very valuable. GemFire provides many options with regards to where and how the cached data gets managed. An architect can choose the appropriate caching architecture on an application by application basis depending on the performance and data volume requirements.

#### *Managing data in Local Application Process (embedded cache):*

In this cache configuration, available for Java, C++ or .NET applications, the cache is maintained locally in the application and shares the space with the application memory space. Given the proximity of the data to the application, this configuration is most appropriate when the cache hit rate is high while offering the best performance. Multiple such embedded caches can be linked together to form a distributed peer-to-peer network as shown in Figure 2.

#### *Hierarchical Caching or Client-server caching*

Hierarchical caching is a deployment model that allows a large number of caches in any number of nodes to be connected to each other in a parent-child relationship. In this configuration, client caches—GemFire caches at the outer level of the hierarchy, or edge—communicate with server caches on backend servers. Server caches can in turn be clients of other server caches and so on. It is a federated approach to caching data for a very large number of consumers and to guarantee cache scalability.

One such hierarchical cache configuration is depicted in Figure 4.

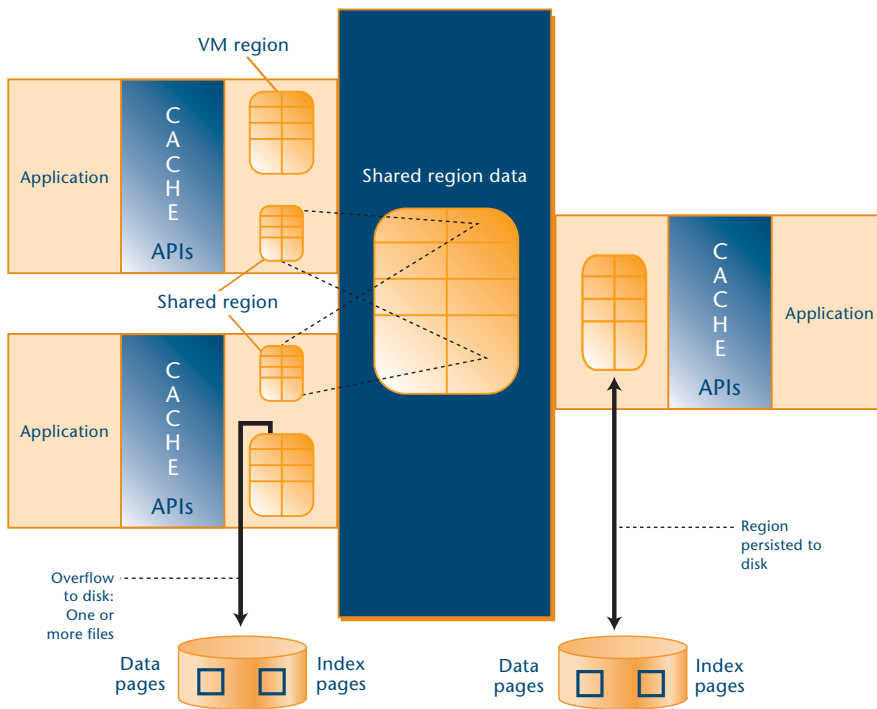


Figure 4: Cache storage models

A miss of the client cache results in the request being delegated to the server cache. A miss on the server will typically result in the data being fetched from the origin data source.

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 inter-cache communications, the traffic to the client caches is dramatically reduced promoting scalability. The server cache is typically deployed on a bigger machine and shields unnecessary traffic to the data source. The hierarchical cache configuration, by design, is loosely coupled, wherein the client cache and the server cache do not participate as part of a single distributed system. In fact, client application deployments themselves could be in a cluster and deployed using a distributed system of their own. The servers likewise can be connected to one another in a single distributed system, mirroring data with each other for high availability and load balancing requests from clients.

Communication between server and client is based on connections created by the client. This allows clients to connect to servers over a firewall. Client caches can create interest lists that identify the subset of data entries for which updates are sent from the server to a particular client. Server to client communication can be made asynchronous, if necessary, via a queuing mechanism with a 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 up to date value for a given region entry. Through this mechanism, it is ensured that the client-side performance does not bottleneck the cache server and impede its ability to scale to a growing number of clients.

### *Partitioned Caching*

The amount of addressable memory is limited to about 4GB (or about 2GB in Linux) in 32-bit operating environments. In addition to this limitation, if the cache is co-located 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 (in memory) much larger than the process space limit or the capacity of a single machine, GemFire supports a model for automatically partitioning the data across many processes, spread across many nodes. GemFire provides this functionality through highly available, concurrent, scalable distributed data structures. Applications simply operate on the region API and behind the scenes; GemFire manages the data across the members of the distributed system, guaranteeing that data access is at most a single network hop. User-defined policies/configurations control the memory management and redundancy (for high availability) behavior of these partitioned regions. By configuring the required level of redundancy, node failures can be automatically handled, as client get/put requests will automatically be redirected to a backup node. When a node holding a data partition fails, the data regions held in it are automatically mirrored to another node to ensure that the desired redundancy levels are maintained.

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 filling memory and/or the rate of data change is high. To deal with unequal quantities of memory allocated across member nodes, the GemFire system automatically initiates re-balancing - the act of moving data buckets from heavily loaded nodes or nodes where the managed data is at or close to the upper bounds, to lightly loaded nodes.

### *Loosely coupled distributed systems for unbounded scalability*

Peer-to-peer clusters are often subject scalability problems due to the inherent tight coupling between cluster members. These scalability problems are exponentially magnified if one considers the scenario involving multiple clusters within a data center or even worse the scenario of data management across multiple data center sites that may be geographically spread out across a WAN. GemFire offers a novel model to address these topologies ranging from a single cluster all the way to multiple data centers across a WAN (shown in Figure 4.a). This model allows distributed systems to potentially scale-out in an unbounded and loosely coupled fashion, without loss of performance and data consistency. At the core of this architecture is a gateway hub/gateway model to connect and configure distributed systems/sites in a loosely coupled fashion. Each GemFire distributed system can assign a process as its gateway hub, which contains multiple gateways that connect to other distributed systems. Backup gateways and gateway hubs can be set up and configured to handle automatic fail-over. Updates that are made in a particular system can be propagated to another system via a queuing mechanism managed by each gateway. The receiving distributed system

sends acknowledgements after the messages have been successfully processed at the other end. In this fashion, data consistency is ensured across data centers that may even be spread out globally. Messages in the gateway queue are processed in batches, which can be size-based or time-based. When messages sent via a gateway are not processed correctly at the receiving end due to the receiver not being available or due to an exception, 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. The 5.0.1 product documentation provides a detailed explanation of the different multi-site configurations possible.

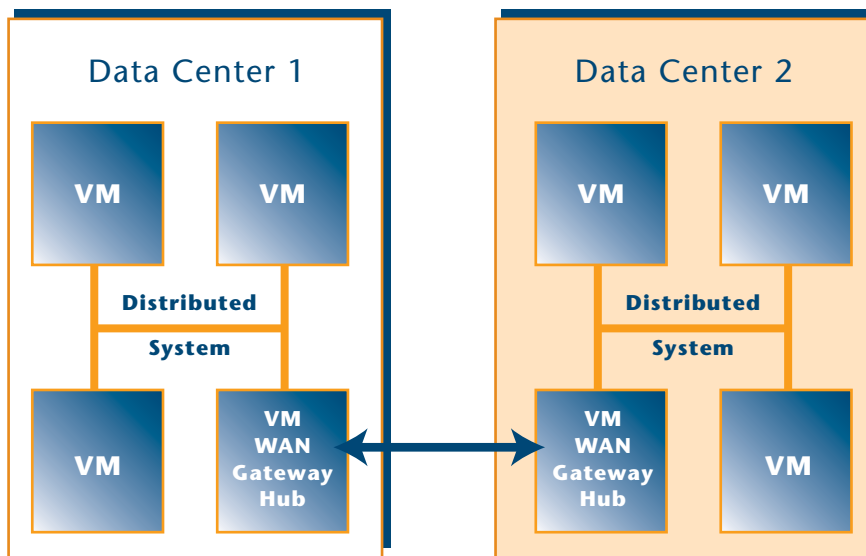


Figure 4.a: Loosely-coupled GemFire deployment across a WAN

*Manage large quantities of data by "overflowing" to disk:*

In this configuration, the cached data that cannot fit into available memory automatically spills over to local disk. GemFire uses an LRU algorithm to move the least recently used objects to disk. This configuration specified as a cache region attribute should be used when dealing with large quantities of data. The disk files are automatically recycled upon cache restart. GemFire optimizes the management of data on disk using Hashed tree indexes.

*Manage data persistently on disk:*

In this configuration, all cache region data is stored persistently on local disk as a backup. GemFire will automatically recover data from the disk files upon restart of the cache. To optimize disk writes, the cached regions can be configured to write all changes to disk at regular intervals rather than synchronously. This option should only be used by applications that can tolerate incomplete recovery upon failure.

### **3.3 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, such as bulk load data from a database 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.

GemFire Enterprise also supports Distributed OQL (D-OQL). With this feature, queries can span data across multiple nodes and across partitions discussed in a prior section.

### **3.4 Transactions**

GemFire Enterprise provides a comprehensive transaction management framework to ensure reliable and consistent data operations. Cache transactions can be applied across cache regions, and can be coordinated either by the cache itself or an external transaction manager, such as the one in a J2EE server. With normal cache operations, GemFire automatically detects the presence of an ongoing JTA transaction and participates using the standard JTA synchronization callbacks. The use of the JTA synchronization call-back avoids unnecessary 2-phase commit operations, while still ensuring transaction atomicity. GemFire Enterprise also includes its own JTA implementation for controlling global transactions in non-J2EE programs. JDBC data sources can be set up in cache configuration files and accessed, as in J2EE, through JNDI. The transaction context is propagated to other members in the GemFire system to provide full consistency between the cache and database.

TRANSACTION\_READ\_COMMITTED isolation level is used for all transactional operations. Applications can choose an optimistic model and use local transactions and propagate the transactional state to other cache members using Distributed\_NO\_ACK scope (described in the next section), or choose a pessimistic model and guarantee changes are applied to all cache nodes in a consistent fashion. GemFire Enterprise's transaction service guarantees ACID properties for all transactions.

## 4. DATA DISTRIBUTION AND CONSISTENCY MODELS

By default, GemFire provides a peer-to-peer distribution model where each cache instance is aware of every other connected instance. 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 is configured at a cache region level. Most of the distributed caching semantics are directly based on the initial JCache specification (JSR 107). This JSR specifies API and semantics for temporary, in-memory caching of Java objects, including object creation, shared access, spooling invalidation, and consistency across Java Virtual Machines.

GemFire extends the JSR 107 initial specification to support various distribution models, additional languages and consistency models for applications to choose from.

### 4.1 Data Consistency Models

#### *Synchronous communication without application acknowledgement*

Applications that do not have very strict consistency requirements and have very low latency requirements should use synchronous communication model without acknowledgements to synchronize data across cache nodes. This is the default distribution model and provides the highest response time and throughput. Though the communication is "out-of-band", the sender cache instance makes every attempt to dispatch messages as soon as possible diminishing the probability of data conflicts.

#### *Synchronous communication with application acknowledgement*

Regions can also be configured to do synchronous messaging with other cache members. With this configuration, the control returns back to the application only after the receiving caches have all acknowledged receipt of the message. This pessimistic mode should be used with prudence especially with increased number of cache members accessing the same region.

#### *Synchronous communication with distributed global locking*

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. 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.

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

- "Replication on demand": Data object is replicated to where it's used. (A 'PULL' model). In this model, the object resides only in the member cache that originally created it. Objects arrive to other cache members only when the connected applications request the object. The object is lazily pulled from other member caches. Once the object arrives, it will automatically receive updates to the object as long as the member cache retains interest in the object.
- "Key replication": Only the keys of the objects cached are replicated - A 'PUSH' model. (This model can preserve network bandwidth and be used for low bandwidth networks)
- "Total replication": All data is replicated (A 'PUSH' model)

#### **4.2 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. 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 in one of the following ways - a.) block any cache operations, b.) allow certain specific cache operations, c.) allow all cache operations or disconnect and reconnect for a specified number of times to check if the required roles are back online.

The role declarations can also be utilized 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 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.

In this fashion, the operational reliability and consistency of the system can be managed as desired 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.

### **4.3 Handling slow and 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 EDF 5.0, a distribution timeout can be associated with each consumer, so that if a producer does not receive message acknowledgements within the timeout period from the consumer, it can switch from the default synchronous communication mode to an asynchronous mode for that consumer. This kind of a switch is primarily used only for regions that support the `synchronous_without_app_acknowledgement` consistency policy. 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. 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. 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 reconnect afresh into the GemFire system, preferably after resolving the issues that caused the consumer to operate slowly. 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 not, 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.

## **5. HIGH 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 co-located 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.

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.

A running GemFire cache system can have up to two essential components that run outside the application process; namely the shared memory segment and the GemFire manager process. For failover purposes, GemFire ensures 100% protection from failures in these out-of-process components by allowing the administrator to configure a "hot-standby" cache. The GemFire clients can be programmed to automatically switch to the standby cache when any un-recoverable error condition is detected. The data in the primary and standby are kept in sync at all times.

## **6. HETEROGENEOUS DATA MANAGEMENT**

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

Given that most IT environments are characterized by more than one application platforms, it is imperative for a distributed caching 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 cache operations available to Java applications via the standard APIs. This kind of data management obviously involves mapping of C++ or .NET objects to Java and vice-versa for storage and retrieval respectively. This mechanism is however transparent to the application programmer once the initial configuration is completed.

## 7. ENTERPRISE CONNECTIVITY

### 7.1 *Connecting with Data Sources*

GemFire provides a simple set of plug-in interfaces for application developers to enable connectivity with remote data sources such as databases, applications, etc. GemFire provides an "out of the box" plug-in for connecting to an enterprise JMS bus. GemFire also provides examples that illustrate how the plug-in interfaces can be used to connect GemFire to a relational database using Hibernate, the popular open source OR mapping tool.

Application developers implement a simple interface called 'CacheLoader' to load data from an external source into a GemFire cache. A loader is automatically executed when an object lookup in the cache results in a miss. GemFire takes care of managing and distributing the object to other cache nodes in accordance with the configured policies. For synchronizing changes to objects in the cache with a data source, the plug-in provides two additional interfaces, 'CacheWriter' and a 'CacheListener'. A 'CacheWriter' enables "write-through" caching and is used to synchronously write changes to the data source before applying the change in the distributed cache. A 'CacheListener' on the other hand, enables "write-behind" caching where the change is first applied to the cache and then asynchronously applied to the data source.

All plug-in implementations can be configured either through the GemFire cache XML configuration or dynamically in the application using APIs. Each loader, writer or listener is associated with a single cache region. GemFire offers flexibility on where cache loaders, writers and listeners are executed. For instance, in a widely distributed environment, the data source may not be accessible from all nodes for security or network topology reasons. A cache miss on a cache that does not have access to the data source will automatically trigger a remote data loader (usually in close proximity to the data source) to retrieve the data. Similarly, writers and listeners can also 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.

### 7.2 *Distributed Event Notification Services*

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.

## 8. SYSTEM MANAGEMENT

### *8.1 Troubleshooting Tracing and Tuning*

GemFire includes a number of tools for debugging, troubleshooting, tracing and tuning.

- **CONSOLE:** GemFire includes a console that allows users to control and monitor all GemFire nodes running on the network from a single machine. Using this GUI tool, each distributed cache can be started/stopped and dynamically configured.
- **LOGGING:** Logging can be turned ON for each cache individually and provides a way to trace the system. GemFire enables logging to be turned ON at various levels ranging from just configuration information to very detailed logging. The logging information generated from each cache can be consolidated to analyze the events across an entire distributed system. GemFire provides a tool (or using the Console) to merge log files allowing the developer to quickly troubleshoot the problem that may span several nodes. The logging system is also exposed via an API for applications to log messages. In addition, there are additional debugging switches for various cache components to provide component specific details.
- **STATISTICS:** In addition to logging, GemFire gathers comprehensive statistical information such as cache hit rate, number of gets, time for various operations, application process level stats, related OS stats, distribution message stats, etc., providing a very fine level of monitoring. The various system statistics are captured in memory and a daemon thread sweeps the stats at configurable intervals and archives these to a "statistics" file on local disk.
- **CHARTING:** The console also provides a special charting tool to monitor, chart and correlate any system statistics gathered by GemFire (See figure 5).
- **INSPECTOR:** The GemFire console includes an Inspection tool that allows the developer to inspect the cache contents at the region level. The fields of individual objects can be viewed.

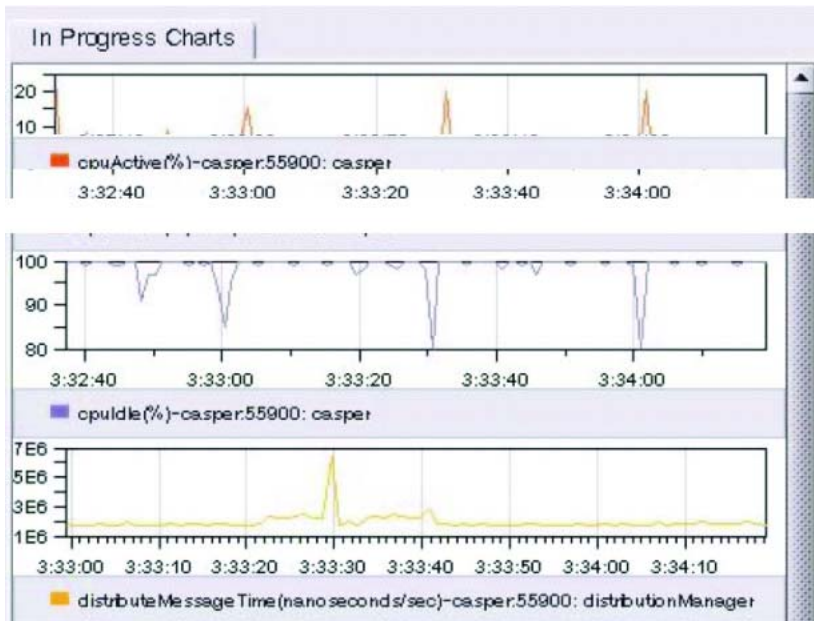


Figure 5: GemFire System Statistics Charts

## 8.2 System Management Tools

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

- 1) The GemFire Console
- 2) JMX APIs and Agent
- 3) 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.

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.

## **9. SECURITY FRAMEWORK**

Almost every enterprise has security requirements and also specific mechanisms and infrastructure to address them. GemFire manages data in many nodes where access to data has to be protected. The security services provided by GemFire have the following characteristics:

*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, and the JMX agent. 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.

*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.

*Confidentiality or 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.

## 10. USE-CASES

### 10.1 GemFire in a program trading environment

A leading Investment banking, securities trading and brokerage firm has deployed the GemFire Trading Reference Architecture as an integral part of their program-trading infrastructure (basket-trading).

#### *Business Problems*

Their business workflow involves receiving bulk orders from institutional clients as well as signals from market data feeds. These orders are broken down into "child" orders/baskets and then routed to the respective exchanges for execution. The trade executions are tracked, orders filled and notifications regarding failed trades are relayed back to the client. The "parent" orders, "child orders" and execution information are persisted in a relational database and retrieved as necessary during the order management process.

*With the existing architecture, the following business problems occur:*

- High Latency issues with an RDBMS based architecture
- Inability to handle the necessary trading volumes to support large orders at the end of trading day
- Lack of Reliable fail-over mechanisms

*Consequences:*

- Loss of revenue from turning down orders/consumption from their own positions
- Lack of reliable mechanisms to ensure business continuity

*The GemFire Solution*

GemFire Enterprise, which is the underlying component of the reference architecture for trading, makes data available on-demand to applications regardless of the underlying data sources or formats. GemFire Enterprise is built on the industry's fastest and most reliable high-performance data distribution and caching system.

Figure 6 illustrates the GemFire based architecture for this program trading environment. The key technical features and services provided by this solution include:

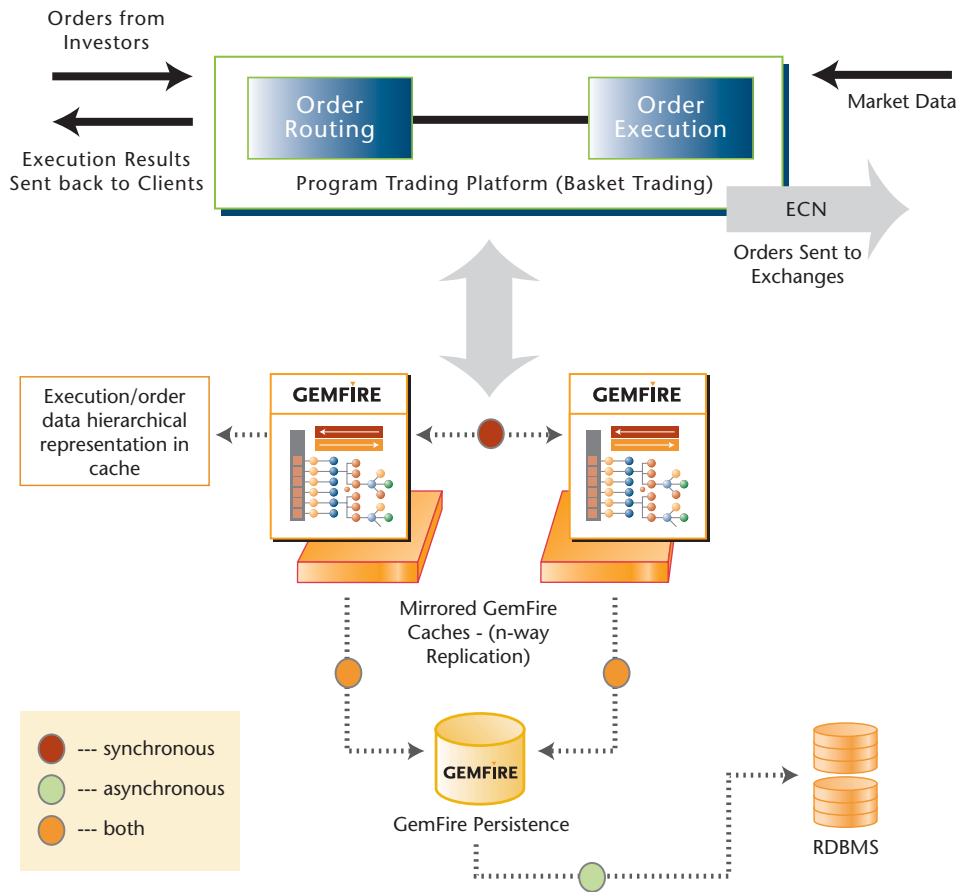


Figure 6: GemFire based Program Trading Architecture

**High Performance Caching:** Order and trade information is held in the distributed GemFire caches that serve multiple order execution engines, which are instantly accessed/updated during the order routing and execution process. Data regions (caches) can be defined based on business logic to hold related data in a single cache instance. Extremely high-speed reads / writes, and consequently high order-processing volumes, are ensured through this mechanism. Also, data latency issues are successfully resolved. New orders that enter the system and order execution data are both recorded on a transaction log with checkpoints that enable recovery from failure (as discussed later).

**Replication:** GemFire data caches are replicated across multiple nodes, with synchronous data propagation to mirror nodes, to ensure 100% data backups. This eliminates single points of failure

and balances client loads. When a failed application restarts, it can automatically reload its cache from a mirror node and seamlessly process client requests. This feature is extremely critical for this program trading environment, where even a few moments of downtime can represent a significant monetary loss. In certain scenarios, a failed application may also recapture its state by recovering data from the transaction log, based on checkpoints that avoid reloading of the entire transaction log.

*Persistence:* In order to ensure complete resilience to system failure and data recovery especially in cases where a significant number of the replicated servers are down, GemFire can be configured to persist the cached order and execution information to disk. As illustrated in Figure 6, this persistence can be configured to be synchronous (as and when the cache is updated) or asynchronous (batch writes to disk). Data stored on disk can automatically be reloaded to a cache on application restart. Persisted data can also be asynchronously archived in a relational database. When an overflow condition is encountered in the in-memory cache, the system can be configured to overflow to disk and dynamically scale to increasing trade volumes.

*Data Distribution:* GemFire enables agile distribution of data across the different components of the order management/trading platform. The distributed caching features of this solution enable a portion of the data to be held in memory within an application and make that immediately available to another application that needs it, either on-demand or in a push-mode. This enables applications and system components to share data effectively.

*Object representation:* Parent orders (aggregate orders), child orders (basket orders sent to exchanges), execution and order fill information are represented in object formats, including their relationships, in the GemFire Enterprise Data Fabric. This enables applications to work with GemFire directly without the need for any complex transformations. Avoiding these transformations further improves the performance of the trading platform.

#### **BENEFITS FROM GEMFIRE:**

- Up to 6000 trades/sec (cache writes with synchronous persistence) and 14000 trades/sec (cache writes with asynchronous persistence) and over 200,000 cache reads/second. This represents approximately a 600% increase in performance
- Ability to handle sporadic loads due to large customer orders and keep their order books open under a minute to market close.
- Increased reliability due to mirrored order/execution data caches and persistence
- Representation and distribution of complex order and trading data in object formats without the need for relational or other formats

#### **10.2: GemFire in an online fixed income securities trading J2EE portal**

One of the largest securities brokerage and banking providers is using GemFire Enterprise Data Fabric (EDF) in a trading portal initiative. This initiative aims to provide a single point of access for trading across a variety of fixed-income instruments like Municipal bonds, Corporate bonds,

Mortgage backed bonds, etc. The ultimate business goal is to enrich the trading experience for the users by providing a wealth of information through a single touch-point and thereby drive customer satisfaction plus increase business volumes.

*The Problem:* The current architecture in this environment includes simplistic in-house caching technologies as an in-memory data storage layer. However there are several issues that exist with this solution with regards to availability, scalability and performance - typical issues that plague any online infrastructure. First, there are issues around extracting, aggregating and caching data from multiple backend databases and applications that store large volumes of securities related information. Secondly, distribution of this data across a farm of application servers in order to service client requests without any latency is problematic. Thirdly, propagating any changes of the underlying data source to the application layer poses its own set of challenges. Finally, there are issues around the scalability of this architecture to cater to increasing user loads without significantly impacting performance. All these requirements highlight the need for a robust data infrastructure. GemFire Enterprise, a key product component of the GemFire EDF, offers distributed caching, data distribution and notification, data virtualization and high availability features for this portal, with easy deployment capabilities.

*The GemFire Solution:* Figure 7 highlights the key aspects of the GemFire based solution architecture. When client requests are channeled to the portal, they are serviced through data stored in the distributed GemFire EDF (spanning several J2EE application server nodes). This fabric seamlessly aggregates data from the different securities databases and applications through the data virtualization framework (data-loaders) and caches them, so that end-user requests are serviced instantaneously. The data stored in the cache can also be transformed into multiple formats to cater to different application needs. This data layer can also serve as a data source proxy and can isolate users from the effects of data source unavailability. Changes to the underlying data are communicated to the GemFire Enterprise layer through XML events on an enterprise message bus resulting in the invalidation of the relevant objects in the cache. Cache data can also be invalidated through configurable timeout policies that GemFire Enterprise supports. Additionally, the mainframe database shown in Figure 7 also has a GemFire cache layer that it uses to service other application clients that request securities data. Through this distributed topology, GemFire provides a pervasive data fabric that aggregates data, stores and caches information and enhances data availability throughout the entire application network thereby providing a scalable model for handling data distribution to service end-user requests efficiently.

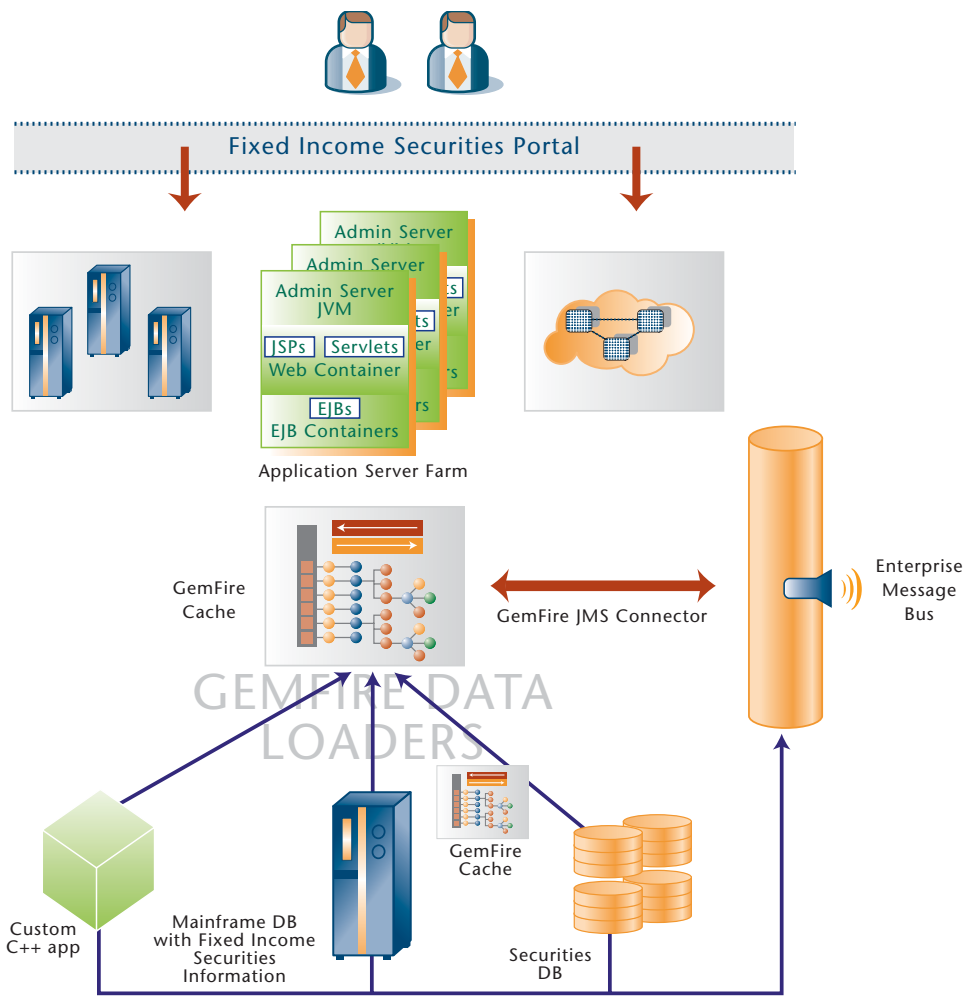


Figure 7: GemFire based architecture for an online portal

**Benefits:**

- High data availability through intelligent data caching
- Unified data access and aggregation across multiple data sources
- Scalability through a distributed caching network that spans multiple application servers
- Data consistency management through notification and data invalidation mechanisms
- Increased resilience to data source unavailability

**Bottom Line:** All these benefits translate to an enriched customer experience anchored on rich content and high performance, which results in improved customer satisfaction, increased customer retention and higher revenues.

### 10.3 GemFire Data Grid for Risk Analytics

The Problem: A leading investment bank was facing a significant problem with their risk computation and analytics grid that was deployed to support their end of day clearing and settlement activities. The cycle time for this process was so long that it almost stretching into the start of the next day's trading cycle. As a result, the operational risk in this environment increased to a great extent. The long cycle time was primarily due to a surge in data volumes that had to be manipulated for risk calculations. Market conditions as well as compliance requirements, both of which required additional sources of information to be included in the risk calculations were the major reasons for this increase in volumes. An analysis of the situation revealed that data latency and data movement accounted for about 70-80% of the risk computation cycle time.

The GemFire Solution: GemFire EDF was deployed as a data grid solution to bolster this risk computation infrastructure. As can be seen in Figure 8, the GemFire deployment involves two data grids - one supporting the pre-processing and theoretical value computation grid and another for the risk calculations. These two grids account for roughly 2 billion calculations across multiple securities and portfolios. From a data flow standpoint, market data snapshots and securities data are pushed into the data grid and held in data regions that are replicated for high availability. These first-level data regions handle large volumes of fast moving data and serve-up relevant subsets of data to a set of 'near' or 'edge' caches (smaller data volumes) that are collocated with the pre-processing compute grid. Such a hierarchical network of caches also supports instantaneous sharing of common data elements as well as intermediate results across multiple applications that repeatedly enrich the data. The results from the pre-process grid are moved to the risk calculation data grid that has a similar structure with a set of replicated large data volume cache servers supporting edge caches that provide local data access to risk applications. GemFire also enables disk persistence of all information held in the in-memory grid, for complete data recovery.

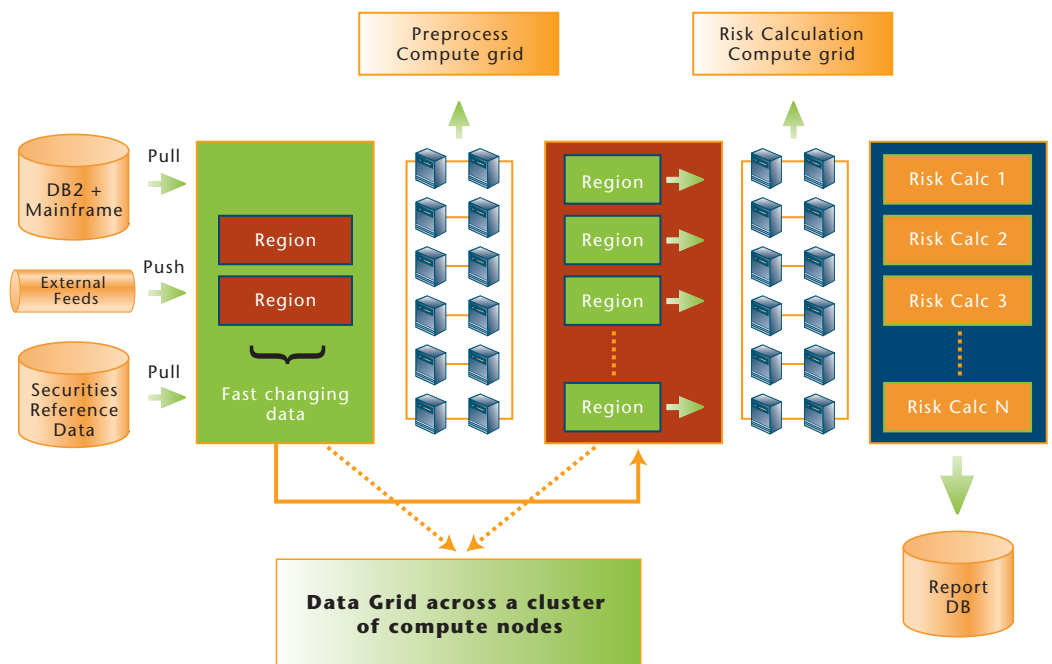


Figure 8: GemFire data grids in a risk computation infrastructure

Through intelligent data caching, distribution and replication mechanisms, the GemFire EDF increases data availability for the risk calculation applications and provides them instant access to relevant information. By positioning the data close to the consumers (risk calculators) through a distributed network of data regions, the data latency issues are successfully resolved.

This infrastructure is deployed on a highly distributed Linux Architecture using RDMA based Infiniband topology. GemFire ideally complements the RDMA by distributing data across interconnected nodes (e.g. blades), creating a single system image of shared data that elegantly spans the system area network created by Infiniband's hardware virtualization technology. By virtue of sharing state in a distributed manner across tightly interconnected blades, GemFire provides massive scale out, business continuity, and load-balancing of data resources.

**Benefits:**

- Reduction in risk computation cycle from 8+ hours to less than 2 hours - ability to run these computations more often (intra-day)
- Access rates of around 350,000 reads/second, which translates improved compute speed
- Increased accuracy through the ability to handle larger data volumes and more diverse data entities that facilitate richer risk calculations.
- Greater scalability (more compute nodes) and high data availability through a replicated, in-memory risk data layer

**Corporate Headquarters:**

1260 NW Waterhouse Ave., Suite 200 Beaverton, OR 97006 | Phone: 503.533.3000 | Fax: 503.629.8556 | info@gemstone.com | www.gemstone.com

**Regional Sales Offices:**

New York | 90 Park Avenue 16th Floor New York, NY 10016 | Phone: 212.786.7328  
Washington D.C. | Phone: 301.325.8405  
2041 Mission College Blvd., Suite 250, Santa Clara, CA 95054

Copyright© 2007 by GemStone Systems, Inc. All rights reserved. GemStone®, GemFire™, and the GemStone logo are trademarks or registered trademarks of GemStone Systems, Inc. Information in this document is subject to change without notice.