Distributed Data Management with VMware vFabric GemFire

Real-Time Data Correlation: Latency and Sustained Operations

TECHNICAL WHITE PAPER

vmware®

Table of Contents

Abstract

VMware® vFabric™ GemFire® is in-memory distributed data management platform that pools memory (along with CPU, network and—optionally—local disk) across multiple processes to manage application objects and behavior. Using dynamic replication and data-partitioning techniques, GemFire offers continuous availability, high performance and linear scalability for data-intensive applications—without compromising data consistency, even under failure conditions. In addition to being a distributed data container, it is an active data-management system that uses an optimized low-latency distribution layer for reliable asynchronous event notifications and guaranteed message delivery.

Two requirements are key for a 7x24x365 high-speed ingest-and-analysis system. The first is to maximize performance by reducing overall system latency. The second is to maintain system operations despite inevitable failures or shortages of RAM, network connectivity and whole systems. This white paper explains how GemFire addresses these requirements.

Terms and Topology

From a topology perspective, GemFire supports several deployment styles. A peer-to-peer topology is the building block for the others (see Figure 1). In a peer-to-peer GemFire system, each member holds one or more sockets open to all the other members. This topology supports the running of application code and data in some or all of the members. As new members join the distributed system, they discover the others through the locator process.

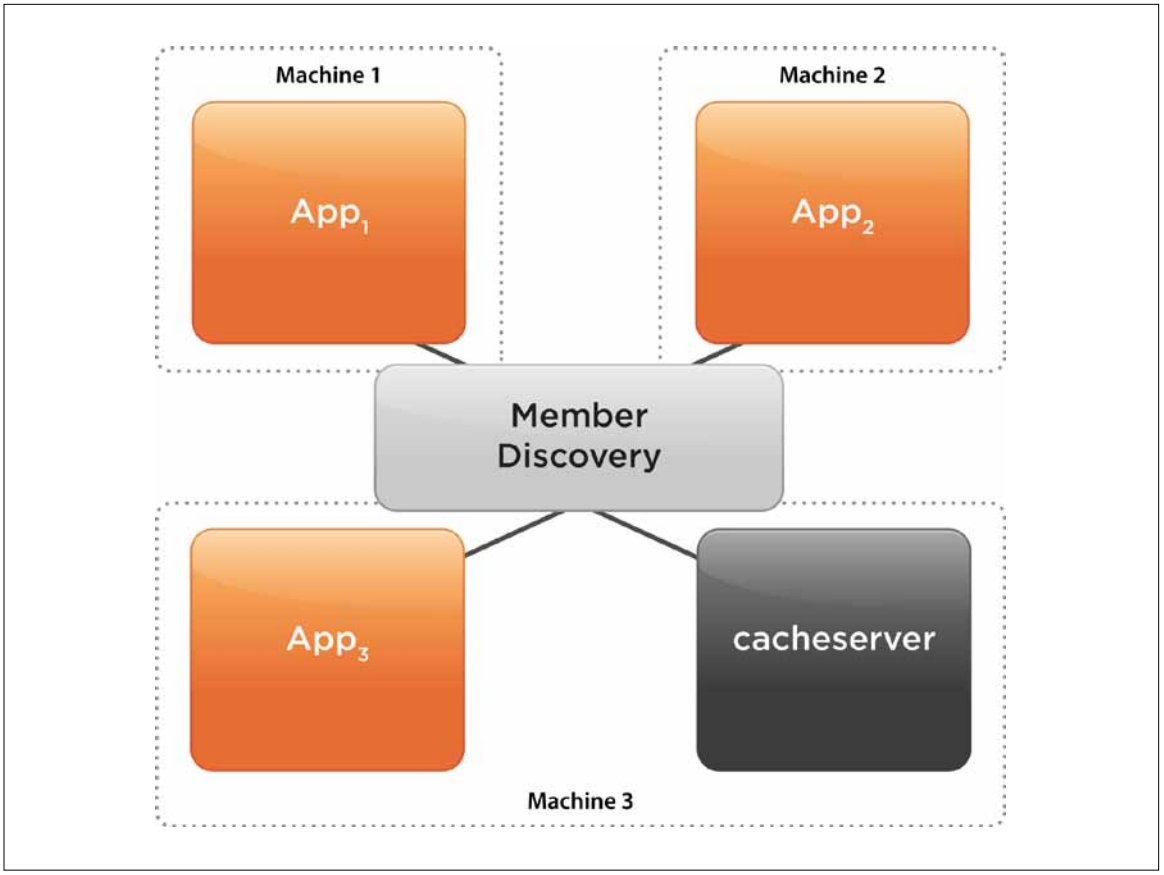

Figure 1. Peer-to-Peer Topology

GemFire also supports a client-server topology in which clients connect to servers in the distributed system (see Figure 2). In this topology, the data resides on the servers, with an optional Level 1 cache on the clients. Application code can run in clients, servers or both.

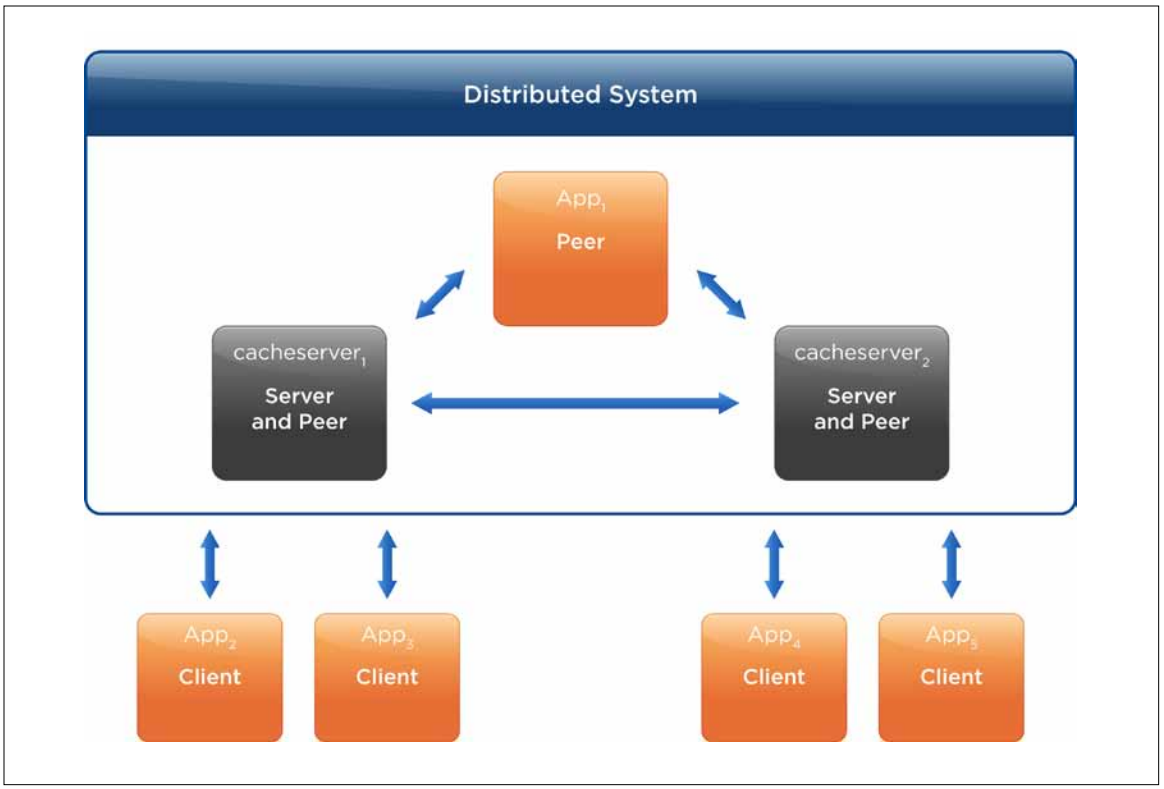

Figure 2. Client/Server Topology

GemFire also supports a multisite topology. In a multisite configuration (generally run across a WAN), distributed systems are configured to communicate with one another through specially configured gateway members (see Figure 3). Each system is its own distinct distributed system, and often each system also acts as a server system in a client/server configuration. The members in each distributed system operate among themselves in standard peer-to-peer fashion. Additionally, the gateway members distribute cache operations to the remote distributed system sites and receive data from them.

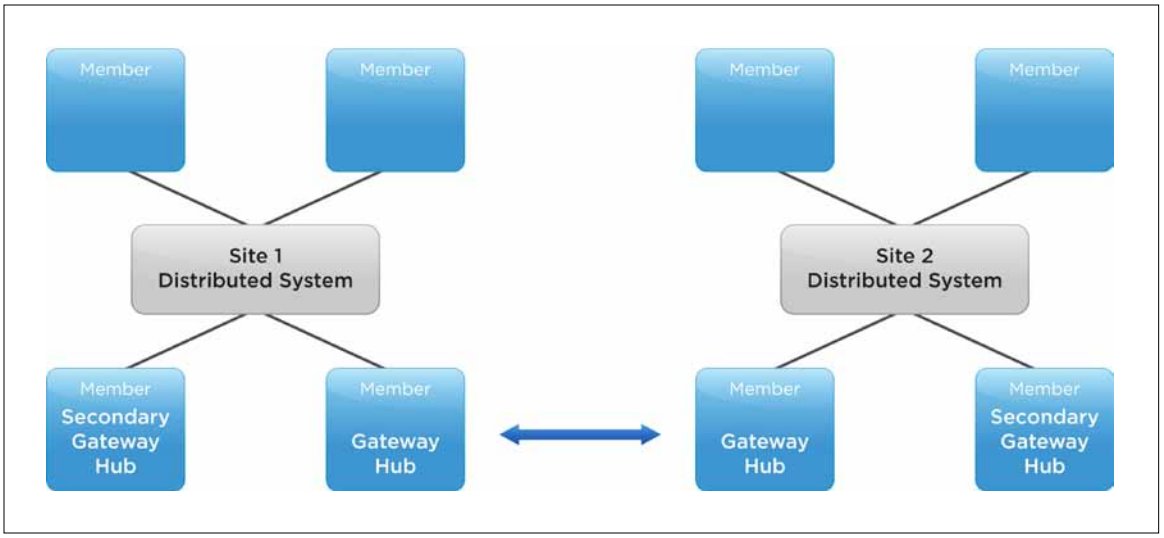

Figure 3. Multi-Site Topology with Wan Connection

A key GemFire concept discussed extensively in this paper is regions. A GemFire data region is a logical grouping within a cache for a single data set. You can define any number of regions within your cache. Each region has its own configurable settings governing elements such as the data-storage model, local data storage and management, data event distribution, and data persistence. A single member can have any or all of four types of regions:

- **Partitioned** – System-wide server setting for the data set. Data is divided into buckets across the members that define the region. For high availability, you configure redundant copies, so that each data bucket is stored in more than one member, with one member holding the primary bucket. GemFire allows additional members to be added to the distributed system and to host another part of the partitioned region. This enables the total size of the partitioned region to grow dynamically.
- **Replicated (distributed)** – Holds all data from the distributed region. The data from the distributed region is copied into the member replica region. Replicated regions can be mixed with nonreplicated ones, with some members holding replicas and some holding nonreplicas.
- **Distributed (not replicated)** – Data is spread across the members that define the region. Each member holds only the data it has expressed interest in.
- **Local (not distributed)** – The region is visible only to the defining member. This region type is typically used by GemFire clients as a communication region to hold server data or messages.

How GemFire Helps Reduce Data Latency

Most data latency in applications results from electronic delay (disk I/O, network I/O) or data impedance mismatch. Electronic delay occurs when data is not in a process. The longer it takes to retrieve the data from the location where it resides, the greater the distance from that location, the more processes the data must pass through and the more transformations the data must undergo, the more latency the requesting application experiences.

GemFire provides a number of mechanisms to avoid these sources of latency. GemFire can enable processes to run in the same process memory as the data. For data-driven applications, GemFire provides cache listeners that enable program logic to run as data arrives in a process in GemFire. This often works well for applications that take in a large amount of data and act on each piece of data as it arrives. For applications that need to retrieve data and then run some code, replicated regions provide a means to manage small (20–30GB) amounts of data in an application process. For larger data sets, GemFire uses partitioned caching to keep the data no more than one network I/O away. Another option for applications that need a large amount of data is to push the business logic to the data using the function services in GemFire. This enables an in-memory, distributed, map-reduce operation to act on the data.

These options are not mutually exclusive. For example, suppose a piece of data arrives that results in a calculation that compares this data with a large set. This could cause the listener to use the function service to execute a function across a partitioned region. That function might need to look at a reference table in the course of the comparison. That reference table could be stored in a replicated region that is co-located with each of the partitions of the partitioned region.

When GemFire writes data into a region, its default behavior is to avoid disk. Disk access is tens of thousands of times slower than memory access—even memory that is one network hop away. GemFire protects data by writing it to two or more computers, either with replicated regions or with replicated buckets (for partitioned regions). To minimize latency, network bandwidth usage and garbage collection, GemFire enables updates to propagate only the changes to an object—rather than the entire object.

Often it is not feasible for code to run in the server processes (for example, when many applications share a set of data at the server level). In this case, the best option is for data to be no more than one network hop away, in main memory of one of the server processes. GemFire always enables the most efficient data access possible given various constraints. When clients connect to a distributed system, they do so region by region. For partitioned regions with n-1/n pieces of data (where n is the number of partitions) data is two network hops away. A GemFire capability called data-aware routing uses more sockets but reduces latency. With data-aware routing, a client request is routed to the exact computer that hosts the partition where that data is stored, minimizing latency. To date, we have not seen another client mechanism from an open-source or commercial off-the-shelf (COTS) vendor that has this capability.

To avoid data latency caused by using the Java Native Interface (JNI) for C, C++ or .NET applications, GemFire provides native client libraries for those languages.

Data Consistency

A data-management platform that runs dynamically across many machines requires a fast, scalable, fault-tolerant distributed system as its foundation. However, distributed systems have unavoidable trade-offs and notoriously complex implementation challenges[i]. For example, the CAP Theorem—introduced by Eric Brewer[ii] and formally proven by Seth Gilbert and Nancy Lynch[iii]—stipulates that it is impossible for a distributed system to be simultaneously consistent, available and partition-tolerant. At any given time, only two of these three desirable properties can be achieved. Hence, distributed systems require design trade-offs. A popular design

i S. Kendall, J. Waldo, A. Wollrath, G. Wyant, "A Note on Distributed Computing", http://research.sun.com/technical—reports/1994/smli_tr-94-29.pdf, 1994. T. Chandra, R. Griesemer, J. Redstone, "Paxos Made Live – An Engineering Perspective", PODC '07: 26th ACM Symposium on Principles of Distributed Computing, 2007.

ii E. Brewer, PODC Keynote, http://www.cs.berkeley.edu/-brewer/cs262b-2004/PODC-keynote.pdf, 2000.

iii S. Gilbert, N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services", ACM SIGACT News, 2002.

approach for scaling Web-oriented applications—driven by awareness of CAP limitations—is to assume that large-scale systems will inevitably encounter network partitions and to relax consistency so that services can remain highly available even when partitions occur[iv]. Rather than fail and interrupt user service upon network outages, many Web user applications (e.g., customer shopping carts, email search, social-network queries) can tolerate stale data and adequately merge conflicts, possibly guided by help from end users once partitions are fixed. Several so-called eventually consistent platforms designed for partition-tolerance and availability—largely inspired by Amazon, Google and Microsoft—are available as products, cloud-based services or even derivative, open-source community projects.

The GemFire team's perspective on distributed-system design is further illuminated by 25 years of experience with customers in the financial-services industry. In stark contrast to Web-oriented applications, financial applications are highly automated, and data consistency is of paramount importance. Eventually consistent solutions are not an option. Financial institutions mandate solutions that are both consistent and available—they want to have their proverbial cake and eat it too. Given CAP *limitations, how is it possible to prioritize consistency and availability yet also manage service interruptions caused by network outages?*

GemFire adopts a shared-nothing scalability architecture in which data is partitioned onto nodes connected into a seamless, resilient fabric capable of spanning processes, machines and geographic boundaries. By connecting more machine nodes, GemFire scales data storage horizontally. Within a data partition, data entries are key/value pairs, with thread based, read-your-writes consistency[v].

Isolation of data into partitions creates a service-oriented pattern whereby related partitions can be grouped into abstractions called *service entities*[vi]. A service entity is deployed on a single machine, where it owns and manages a discrete collection of data—a holistic chunk of the overall business schema. Hence, multiple data entries co-located within a service entity can be queried and updated in a transactional manner, independently of data within another service entity.

Transactions and Sharding

The ability to partition or shard data is a scaling and data-consistency issue, not a latency issue. GemFire provides a number of mechanisms that enable data consistency in a partitioned or replicated system. GemFire partitioning provides for dynamic growth, letting you add new machines to host elements of the partition. GemFire also allows the region to be dynamically rebalanced. To provide maximum flexibility, GemFire has its own hash-based sharding algorithm that supports range-based partitioning. It also allows developers to insert their own algorithm so they can more tightly control which data is on which computer. Many other open-source or COTS partitioning systems either do not allow for dynamic system growth or fail to provide a means for developers to control data placement.

To enable high availability for a data element in a partition, GemFire lets you specify a number of replicated copies of the data that will be provisioned on different computers. To provide for read/write consistency of those copies, one of them is designated the primary. Unlike most other systems that allow for partitioned read/write consistency, GemFire allows client applications to see and use the secondary copies of that data. Data reads are directed to any one of the copies, thereby helping to ensure scalability and performance. Writes are directed through the primary copy only. A write on the primary copy results in a blocking update to all of the secondary copies.

GemFire has a built-in transaction coordinator. GemFire supports transactions on data entities within a partition to enable updates to multiple related objects to be handled in an ACID (atomicity, consistency, isolation, durability) manner. Transactions can span partitioned, co-located regions and replicated regions. Many other open-source and COTS products do not support transactions, or they rely on the slow and cumbersome two-phase commit approach.

---

iv W. Vogels, "Eventually Consistent", ACM Queue, 2008.

v W. Vogels, "Eventually Consistent", ACM Queue, 2008.

vi P. Helland, "Life beyond Distributed Transactions: an Apostate's Opinion", CIDR 2007, http://www-db.cs.wisc.edu/cidr/cidr2007/papers/cidr07p15.pdf, 2007.

Data Subscriptions

To give a client application a consistent view of the distributed system, GemFire provides several ways that clients can subscribe to data in a distributed system. In many distributed caching or in-memory database management system (DBMS) style systems (MongoDB, Terracotta, etc.), if data changes on the server the client is unaware of those changes. Some systems let you know if data you currently have has changed. Only a few systems (all COTS) let you get an initial image and register interest by pattern or query in a "potential" data set of interest. And only GemFire lets you do all of this in a fault-tolerant manner.

GemFire clients can be notified of all changes to data retrieved from the server side. Clients can also get an initial result set and register interest in data by keys, by regular expressions on keys (where those keys must be strings), or by creating a Continuous Query using language statements on the data. GemFire uses the Object Data Management Group (ODMG) Object Query Language (OQL)—a subset of SQL-92 with some object extensions—to query and register interest in data stored in GemFire. Furthermore, in the event of a computer failure in the distributed systems (the server side), GemFire automatically reconnects clients to an alternate distributed system member that has a redundant copy of their registered interest.

Sustained Operation in the Event of Machine, Network or Memory Failure

In addition to low latency, GemFire's other main goals are consistency and availability. The main enemy of availability is the inevitable failure of computers and networks. For distributed-memory solutions, memory shortages can be an additional challenge.

Machine Failure

For distributed-memory–based solutions, the first line of defense against the failure of a single computer is to keep a secondary (tertiary, etc.) copy of the data on another computer. GemFire provides two ways to do this. Regions can be replicated so that a full copy of the data set is on multiple computers. Or regions can be partitioned with a configurable number of read-only secondary copies.

Site Failure

In the event of total power failure at a site, no local data-replication scheme can protect all data in memory from site-level loss. GemFire provides two options to help deal with this issue. On a per-region basis, GemFire provides for optional disk persistence. GemFire has a high-performance round-robin transaction log that is parallelized for partitioned caches. This log is used in conjunction with background processes that write the data onto the disk partition. With local disk at each computer, we generally see that enabling disk persistence degrades throughput performance by only 5-7 percent. Upon restart, startup occurs with automatic initialization from the most-recent data on disk, so you do not need to track which members have the most-recent data. GemFire also includes all of the start utilities you might expect from a DBMS: command-line management of disk file verification, compaction, backup, and detection of missing or corrupt files.

When continuity of operations is critical, persistence to local disk is not enough. A data-replication capability in GemFire—called the WAN Gateway—enables GemFire, on a per-region basis, to replicate data from one distributed system to another. This capability supports multiple sites concurrently. It is a highly available architecture with no single point of failure. Data is queued at the Gateway and sent in optimized batches to all of the other Gateways that are interested in that data region. The queues are memory-based with optional overflow or persistence to disk. In the event of a network failure, the data queues up until the network recovers.

It is possible to specify multiple network paths for the data. It is also possible to enable conflation in the queues, which causes only the last change to any object to be propagated over the WAN. If network bandwidth is limited—or the network is down long enough that the amount of data in the queues would take a long time to send—conflation can seriously reduce the total amount of data sent. The WAN Gateway also supports GemFire delta propagation as discussed in the section on latency. The WAN Gateway can be deployed in active/passive architectures whereby each site owns its subset of data, and in active/active architectures.

Memory Issues

GemFire provides several mechanisms for managing the amount of memory used for data management—and for alerting administrators when the system will need more memory to maintain an increasing amount of data. First, you can specify the total amount of memory a region will use on a system member. This value can be an absolute number or a percentage of the available Java heap. Second, as memory reaches this predefined point, you can (optionally) specify an eviction action, which either removes the least recently used object from memory or moves it to a disk overflow area. Items in disk overflow do take up some memory, because the key and a disk pointer remain in memory. Third, for partitioned regions, you can scale the system to add more memory at runtime by adding more system members. If actively monitoring memory usage though GemFire administration APIs, IT staff has ample advance indication of the need to add GemFire instances to accommodate a growing data load.

Resource Manager

To further help with memory-management issues, GemFire includes a unique resource manager that lets you control memory use in a configurable manner and alert administrators as memory use reaches certain thresholds. The GemFire resource manager uses a garbage-collection (GC) mechanism to control heap use and protect your Java virtual machine (JVM) from hangs and crashes that result from memory overload. The manager prevents the cache from consuming too much memory by evicting old data and—if the collector is unable to keep up—by refusing additions to the cache until the collector has freed an adequate amount of memory.

The resource manager has two threshold settings (both disabled by default), each expressed as a percentage of the total tenured heap (see Figure 4):

- **Eviction threshold** – Above this threshold, the resource manager orders evictions for all regions with eviction attributes set to lru-heap-percentage. This prompts dedicated evictions that are independent of any application threads, and it tells all application threads adding data to the regions to evict at least as much data as they add. The JVM garbage collector removes the evicted data, reducing heap use. The evictions continue until the manager determines that heap use is again below the eviction threshold.
- **Critical threshold** – Above this threshold, all activity that might add data to the cache is refused. This threshold is set above the eviction threshold and is intended to allow the eviction and GC work to catch up. This JVM, all other JVMs in the distributed system and all clients of the system receive a LowMemoryException for operations that would add to this critical member's heap consumption. Activities that fetch or reduce data are allowed.

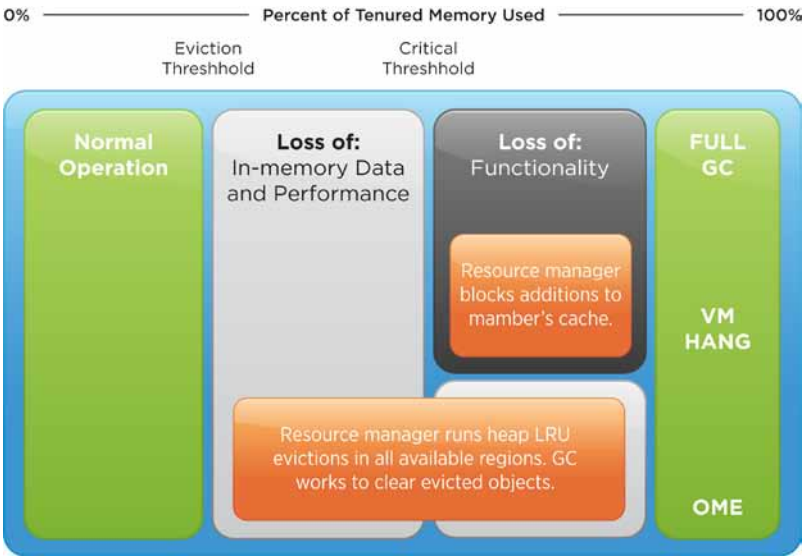

Figure 4. Tenured Memory Thresholds

Handling Network Splits and Outages with Split-Brain Detection

Most distributed data systems do not handle the issue of network splits. The unique (and optional) split-brain detection capabilities of GemFire can keep distributed systems from splitting into two separate running systems when members lose the ability to see each other (see Figure 5). The typical cause of this problem is a network failure. During a network failure, or when partitioning occurs, the problem can result in data inconsistencies or a forced disconnect.

The solution to this problem is to stop one of the two subgroups from continuing to operate independently. Handling of network outages is based on the participation of a lead member and a group management coordinator (the locator). The coordinator is a member that manages the entry and exit of other members of the distributed system. With network partition detection, the coordinator is always a GemFire locator. The lead member is always the oldest member of the distributed system that does not have a locator running in the same JVM and is not using the administrator interface. Two situations cause GemFire to declare a network-partitioning condition:

- If both a group coordinator and the lead member abnormally leave the distributed system within a configurable period of time, the caches of members who are unable to see the locator and the lead member are immediately closed and disconnected. Only abnormal loss of the locator and lead member causes GemFire to declare a network partition. If a lead member's distributed system is disconnected normally, GemFire automatically elects a new one and continues to operate. If a locator is disconnected, a secondary locator takes over.
- If no locator can be contacted by a member, the member closes its cache and disconnects from the distributed system. Because only locators can make membership decisions, a member that cannot contact any locator cannot know if it is isolated from the lead member.

Network-partitioning handling allows only one subgroup to form and survive. The distributed systems are disconnected, and the caches of other subgroups are closed. When a shutdown occurs, alerts are generated through the GemFire logging system, explaining to administrators what action, if any, to take.

For the algorithm based on choosing a single network group to survive to work, it is critically important to control system startup such that the lead member and the locator do not

- Reside on different sides of a crucial shared network component (such as a switch) that could leave each of them on one side of the partition along with some number of other GemFire peers.
- Reside on the same host, in which case a network-adapter failure would shut down the entire cluster.

The way to make the algorithm work is to place half of your capacity on one side of the components you're concerned about (those whose failure would cause a network partition—such as a major switch, or a MAN link between primary and secondary data centers), and the other half on the other side. The placement of the two "special" members makes the surviving side a deterministic outcome (see Figures 5 and 6).

Network Failure—Network Partition Configurations

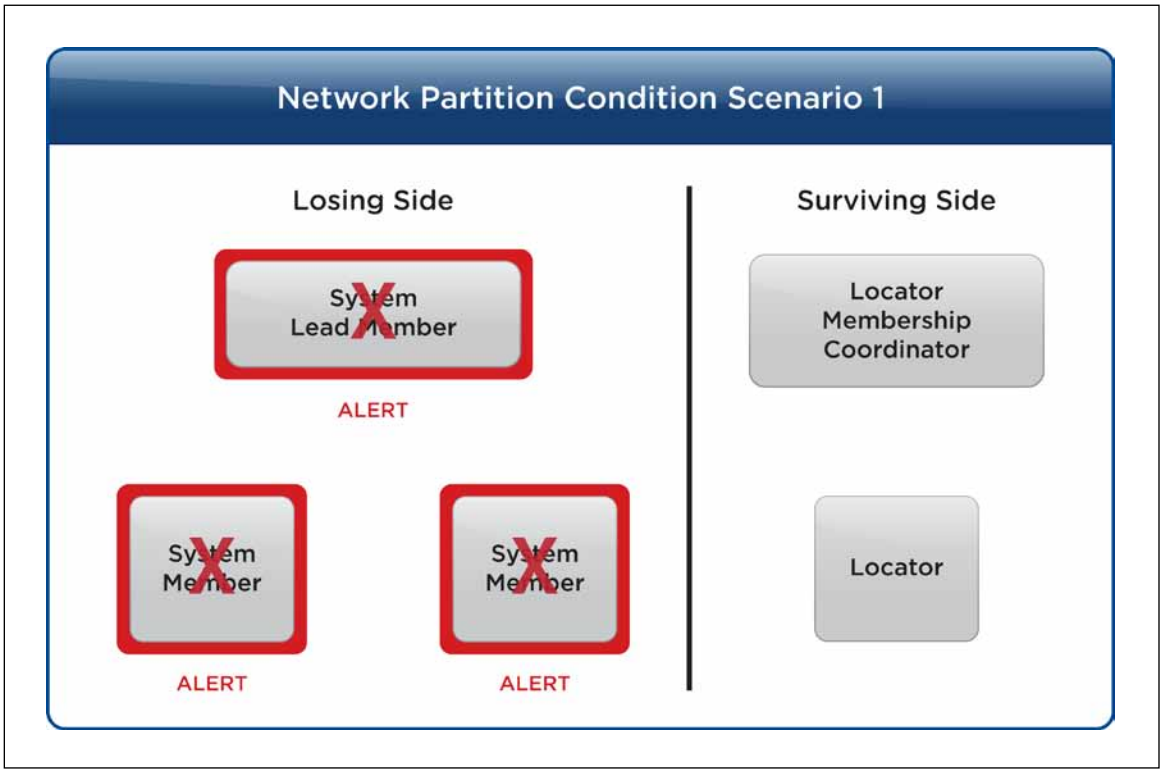

Figure 5. Network Partition Condition - Scenario 1

1. Locators see the lead member leave and remain active.

2. Members lose sight of all locators and are closed because there is no one to make membership decisions. Each member generates an alert. They cannot tell whether the lead member is alive and do not know who the current lead member is.

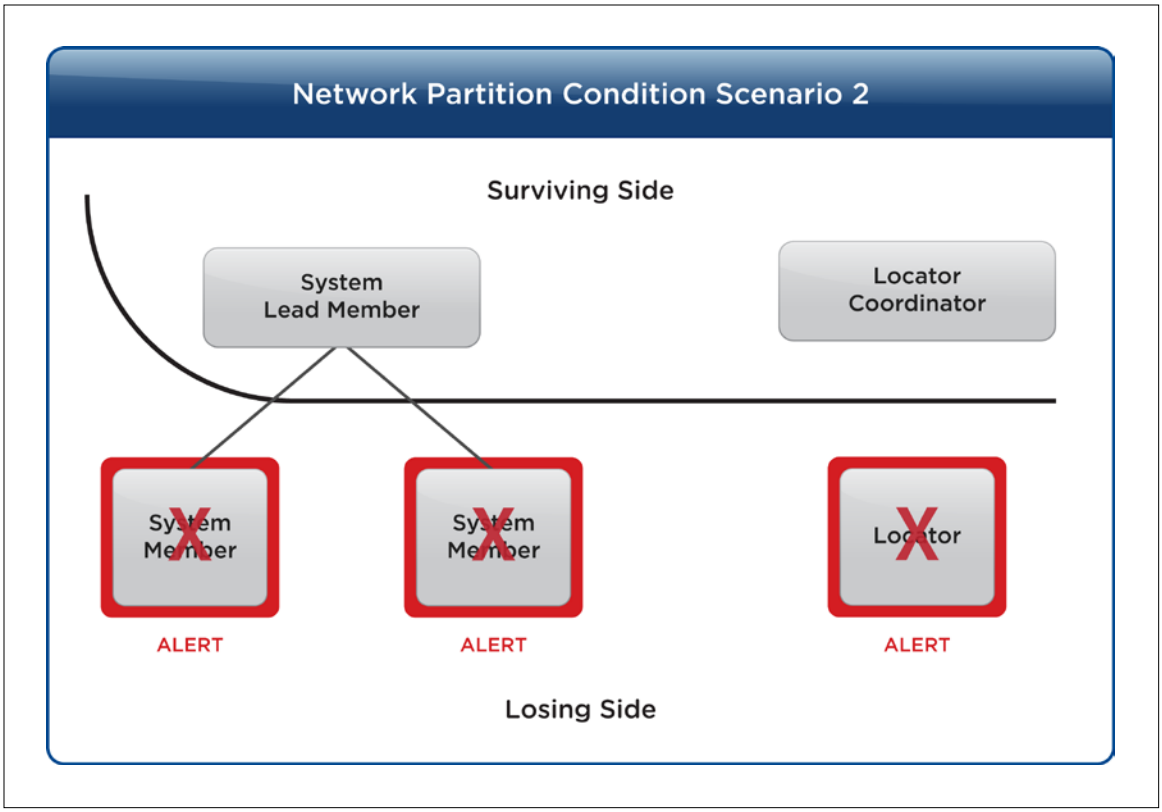

Figure 6. Network Partition Condition - Scenario 2

1. Failure detection uses member-timeout to remove lost members.
2. On the losing side, each member sees the loss of the coordinator and lead member and shuts down, causing each member and the locator to generate an alert.
3. On the surviving side, members still see the lead member and locator, and remain active 120.

Surviving Side

GemFire can notify system administrators or others of the system's condition. The group coordinator (locator) JVM on the surviving side can have an AlertListener configured. Members on the surviving side can have SystemMembershipListener configured to process memberCrashedEvents for the peer system memberson the losing side.

Losing Side

A member that detects a network partition—and does not determine it is on the surviving side—disconnects its distributed systems and closes its cache. If a network partition caused the loss, the processes in the other partition (eligible coordinator, lead member and processes still able to see them) continue to run, electing a new coordinator if necessary. Any clients of the system are unaffected, although they can reconnect automatically to a member on the surviving side.

Pipeline Processing

An alternative to split-brain detection for applications that have subsets of computers that can run as independent groups is to use the membership role capability in GemFire. This capability also handles issues of system relationships between groups of computers in areas such as pipeline data processing.

GemFire Membership Roles

GemFire membership roles give you the ability to decide under what circumstances a distributed system can continue reliable operation after a disruption such as a network failure. A membership role describes how a system member relates to other members, or what purpose it fulfills.

Members can establish interdependent relationships among themselves by playing one or more membership roles. Membership roles are optional, so they are not required for every member in the system. You configure membership roles at the member level and the region level. First, identify all the roles your system members play. Then decide which roles a given member fills, and add those roles to the member's system-level configuration. Typical role names are Producer, Consumer and Backup. When the member connects to the distributed system, it declares those roles.

Then, for each region, decide which of those membership roles must be present somewhere in the distributed system to enable reliable access to that region. You configure the region's role requirements and appropriate actions to take when the distributed system loses or regains one of those required roles. With membership roles, you can configure a cache so that one or more of its regions requires another system member to be present in order for the region to be used reliably. You can set up intermember dependencies so that processing can be halted or altered when dependencies are not satisfied. A member can play multiple roles, and different members can declare that they play the same role.

You can specify membership roles to help you identify processes and the domain functions that a process performs. GemFire exposes APIs to allow systems to see the roles they play or administrators to see which systems are running with which roles. You also can specify actions for when a member with a given role is unavailable to another system that requires such a role be present.

loss-action

The loss-action setting specifies how access to the region is affected when one or more of the roles specified in the region's membership attributes are offline and no longer present in the system membership. Table 1 defines the options for loss-action.

SPECIFICATION (DEFAULT IN BOLD)	DEFINITION
full-access	Access to the region is unaffected when required roles are missing.
limited-access	Only local access to the region is allowed when required roles are missing. All distributed write operations on the region throw a **RegionAccessException** while any required roles are absent. Reads that result in a **netSearch** behave normally, and any attempt to load is not allowed.
no-access	The region is unavailable when required roles are missing. All operations including read and write access are denied. All read and write operations on the region result in **RegionAccessException** while any required roles are absent. Basic administration of the region is allowed, including close and **localDestroyRegion**. *no-access is the default loss-action only if you define membership roles. Without roles, full access to the region is always allowed.*
reconnect	Loss of required roles causes the entire cache to be closed. In addition, this process disconnects from the distributed system and then reconnects. Attempting to use any existing references to the regions or cache throws a **CacheClosedException**. *This setting requires a full cache configuration in the cache.xml file. All programmatically defined cache and region attributes are lost if they are not also in the .xml file.*

Table 1. Options for loss-action

Slow Receivers (Order, Latency)
Notifying clients about changed data and maintaining data ordering are challenges for other distributed data management solutions that support client-side interfaces. Any JSR 107/JCache system—such as Ehcache, Terracotta or Coherence—faces the problem of what to do when a client is slow to receive data. Causes of this condition include JVM pauses, network-adapter issues and high data flow that the system cannot keep up with. Systems other than GemFire either disconnect such clients from the system or slow all clients down to the speed of the slowest client. GemFire has an alternative model that dynamically queues data to such clients on a client-by-client basis.

GemFire can be configured to replicate those queues onto multiple computers so that in the event of server failure, clients can be automatically connected to an alternative server that has their queues. This enables clients to continue processing without even being aware of a server failure. To ensure that no data being sent to a client is lost, GemFire has a pessimistic model for queue replication. This can result in a small amount of data replay on the client. However, GemFire data processing includes a monotonically increasing message number. So GemFire detects duplicate data and discards it before a client application sees it.

Administrative Functionality
GemFire has a full set of Java Management Extensions (JMX)-enabled APIs for management, monitoring and system health. GemFire also has a retrospective-analysis capability to help analyze the detailed low-level statistics that GemFire can be configured to produce.

GemFire has a plug-in for Hyperic®, another VMware vFabric Cloud Application Platform product. GemFire also includes GemFire Monitoring (GFMon) for examining the state of a GemFire distributed system in real time. Figures 7, 8 and 9 show some GFMon capabilities.

A GemFire distributed system produces logs for applications, cache servers and locators:

- **Applications and cache servers** – To create log files, you must set the log-file attribute in the process's gemfire.properties file. Otherwise, the messages go to stdout. These log files can be placed anywhere that is convenient for monitoring. For applications, these log files contain entries from GemFire operation only.
- **Locator** – The locator always creates a log file in its working directory. This logging is not configurable.

Additionally, you can use the JMX Agent to integrate GemFire into any standards-based system-management tool to perform the following management tasks:

- View the distributed system and its settings
- View distributed system members
- View and modify configuration attributes
- View runtime system and application statistics
- View a cache region and its attributes and statistics
- Monitor the health of a GemFire Enterprise system and its components

GemFire even has a command-line interface for handling basic management tasks such starting locators and cache servers, and merging logs. Gfsh (pronounced "g-fish") is a GemFire command-line tool for browsing and editing data stored in GemFire data fabrics. Its rich set of UNIX-flavored commands allows you to easily access data, monitor fabric peers, redirect outputs to files, run batch scripts, execute custom functions and much more.

Figure 7. Managing GemFire with GFMon

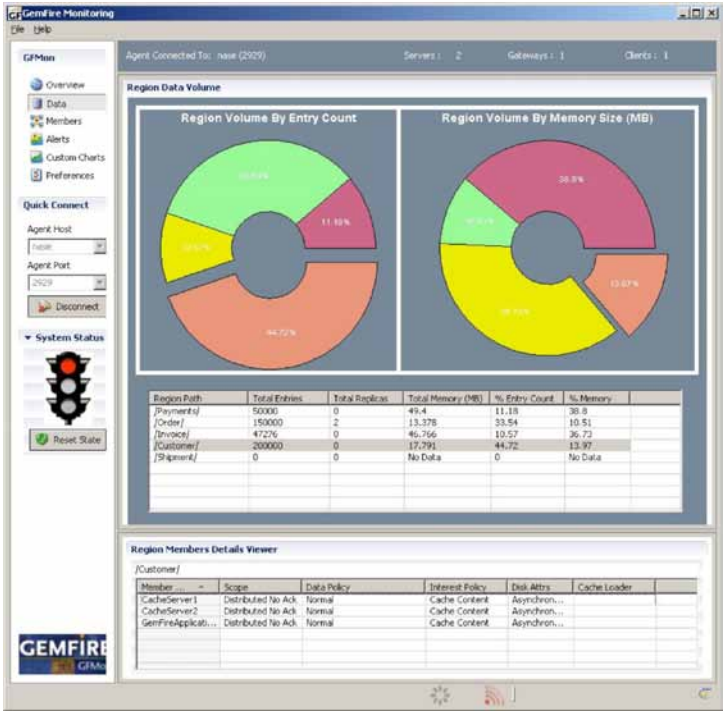

Figure 8. DataBrowser in GRMon

Use DataBrowser to browse the data in a GemFire Enterprise cache server by running ad hoc OQL queries and to monitor real-time changes to a data region by registering a continuous query.

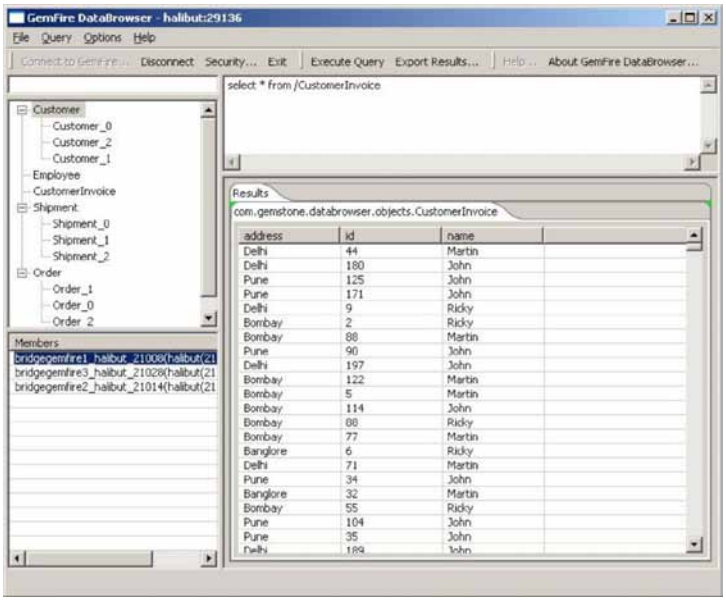

Figure 9. Running Ad Hoc Queries in GFMon

GemFire Administrative API

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, the administration API provides interfaces for issuing and handling system member alerts and for monitoring statistics. You can receive notifications in an application for all membership events, cache and region creation, and log messages.

The health-monitoring API allows you to configure and monitor system health indicators for GemFire distributed systems and their components. There are three health levels of health:

- Good health indicates that all GemFire components are behaving reasonably,
- Okay health indicates that one or more GemFire components are slightly unhealthy and may need some attention.
- Poor health 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. GemFire provides methods for configuring the health of the distributed system and members that host cache instances. Health can be configured on both a global and per-machine basis. GemFire also allows you to configure how often a GemFire health evaluation is conducted. The health-administration APIs make it possible 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.

Retrospective Analysis

The Visual Statistics Display (VSD) tool in GemFire enables you to examine all of the statistics GemFire generates and to correlate them for retrospective analysis.

In VSD statistics chart shown in Figure 10, the manager's evictions and the JVM's GC efforts are good enough to keep heap use very close to the eviction threshold. The eviction threshold could be increased to a setting closer to the critical threshold, allowing the JVM to keep more data in tenured memory without the risk of overwhelming the JVM. This chart also shows the blocks of times when the manager was running cache evictions.

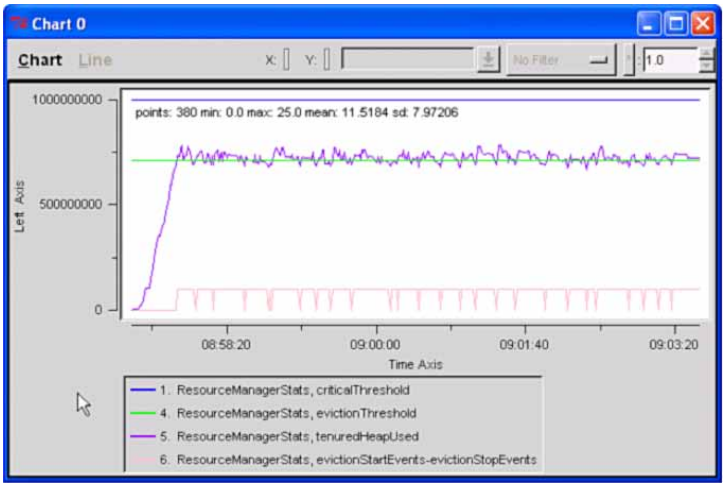

Figure 10. Analyze GemFire-Generated Statistics with the Visual Statistics Display (VSD) Tool

Network-Related Features

Port Use Configuration

GemFire offers product-wide port use configuration, which means you can configure every port that the product uses. This makes GemFire easy to manage and more security-friendly. In the online product documentation, see the new membership-port-range GemFire property in the chapter, "Configuring the System" in the GemFire Enterprise System Administrator's Guide. GemFire supports multiple network protocols on a per-region basis, including the use of ONLY TCP/IP, and the use of unicast or multicast.

Socket Control

GemFire provides a number of controls for sockets and network buffers. You can optimize GemFire throughput by having GemFire use a pair of sockets for each thread it generates when communicating with other system members. Or you can have GemFire use a single pair for each member. In the latter case, some context switching between threads will occur to send and receive data to or from other members. You can limit the number of network connections

IPv4 and IPv6 Support

GemFire is completely IPv6-capable. It also supports mixed IPv4/IPv6 networks, although all system members must use one or the other. Many other products in this space are not yet IPv6-capable.

Disk-Related Features

Auto-Compaction of Operations Log

To avoid the issues that many DBMSs have where a full transaction log can cause a system halt, the Ops Log in GemFire has automatic compaction. Additionally, it can automatically create rollover versions with the total number of versions preserved and total disk space used, completely configurable.

Handling Catastrophic Loss of Storage Disk Data

GemFire provides a way to start a partitioned cache from disk, even if you cannot recover a missing a disk store. You can revoke it from the system during startup so the other members can start. You revoke a disk store by telling online members that a missing member's disk store is no longer the most recent.

Online Backup

The GemFire backup creates a backup of disk stores for all members running in the distributed system when the backup command is invoked. The backup works by passing commands to the running system members. Each member with persistent data creates a backup of its own configuration and disk stores. The backup does not block any activities in the distributed system, but it does use resources.

Conclusion

Compared to other commercial or open-source offerings, GemFire provides the best data-management solution for high performance, low latency and high reliability even on intermittently unavailable network and computer-hardware platforms.

Find Out More

For information or to purchase VMware products, call 877-4-VMWARE, visit www.vmware.com/go/gemfire, or search online for an authorized reseller. For detailed specifications and requirements, refer to the product documentation.

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com