GemFire Enterprise : Best Practices and Capacity Planning guide

This page last changed on Jun 04, 2009 by smartin.

Capacity Planning Process - Executive Summary:

This article provides guidelines and describes the process followed by GemFire solution architects to estimate the resource requirements in terms of memory, number of nodes, network bandwidth, etc to meet the application performance, scalability, and availability goals.

This article is meant to only offer guidelines and assumes a basic understanding of GemFire. While no two applications or use cases are exactly alike, these guidelines should be a solid starting point, based on real-world experience. Much like with physical database design, ultimately the right configuration and physical topology for deployment is based on the performance requirements, application data access characteristics, and resource constraints (i.e., memory, CPU, network bandwidth) in the operating environment.

Capacity Planning Process - Step by Step:

1. Understand your application requirements

Here the GemFire solution architect will qualify and quantify what the current and future data access patterns are so he/she can better make resource provisions to meet the required SLA (Service Level Agreement) or QoS (Quality of Service). With enough planning, some fine tunning, and some validation testing, your GemFire solution can drastically minimize and resolve your data bottlenecks including availability limitations, and, therefore guarantee performance and business continuity.

Knowing about your data access patterns (e.g., operation complexity and concurrency and data sizes and data types) and knowing about your available resources (e.g., RAM, CPUs/cores, bandwidth) will help you to plan for, fine tune, and match your application performance with your SLA requirements.

a) Data:

Here the developer or architect quantifies data types/sizes, lifetime as well as qualifying consumer/producer process access patterns that affect how much data is required overtime. Below are some questions you should ask yourself:

- **How much total data to store in the cache?** (e.g., a few GBs, 100s of GBs)
- **What types of data and how big those cache entries are?** For example:
  - 1 or 2KB Java, C++, C#/.NET objects
  - 5KB files
- **How much data per data type?**
  - It would be helpful to have a base knowledge of the size per data type and how operation frequency against a particular data type and data set.
- **What data dependencies?**
  - Are you objects simple or nested?
- **What are the lifetime requirements for cached data (static vs. dynamic data)?** In other words, are your different cache data entries type long lived and/or very short lived? It is important to understand:
  - How many days worth of data you need to store. You might have different lifetime requirements for different types of cached data entries (e.g., trades, positions, reference, and indicative data).
  - How often the cache might be loaded/flushed from external data sources? It is important to set up the appropriate plans to manage your data loading/flushing and scalability needs, and, knowing your data pattern workflow over time will help you plan for normal system operation. Please remember you can always take advantage of GemFire's dynamic provisioning capabilities to proactively respond to future data and operation load growth.

    As a general rule of thumb, when the architect collects enough information on the size and types of data, and, on the complexity and concurrency of operations, the architect will more precisely be able to make design decissions to build the right solution (e.g., replicated regions for small not-too-dynamic data subsets that fit in one process vs. partitioned regions for small very-fast-changing data sets or very large data sets). If your application calls for CPU-intensive calculations, then you might want to store your data in partition regions to benefit from parallel execution of operations.

b) Operation and client load:

Here the solution architect will more precisely qualify the data access pattern (type and concurrency of operations) as well as start to determine what resources might be available (e.g., aggregate network bandwidth, aggreate client count and operation complexity and concurrency. Below are some questions for the solution architect to review:

- **How many concurrent clients** accessing your cache? (i.e., a handful, hundreds, thousands)
    - Most small and medium size caches will require just one Cache Server cluster (a.k.a., Distributed System). This is true as data sizes, and, operation type and operation concurrency can be perfectly managed by one cache server cluster. This is generally true when the number of clients is small to a very few hundreds.
    - You might have cache clients connected and talking with one or more Cache Server clusters where each Cache Server cluster or Distributed System might contain different types or sets of data.
- **Aggregate cache server to cache client bandwidth:**
    - *Are your publisher/consumer cache clients going to run on a slower network than the cache servers? What bandwidth and how many client sub-networks?
    - Important thing here is understand and provision the appropriate aggregate Cache Server to Cache Client throughput that you are going to require for normal and peak load operations.
    - If running your Cache Servers on Linux, you might want to use two or more 1GbE NIC cards and configured them using Linux bonding mode number 6 (which effectively provides one IP address and the aggregation of all the NICs incoming and outgoing network bandwidth).

- **Data access patterns: What kind and what frequency of operations** are these publisher/consumer clients performing? Important thing here is to have an understanding of the type, concurrency, and complexity of the operations against the cache servers to properly provision based on normal production operation and over-the-time use GemFire's dynamic provisioning capabilities to proactively respond to future more demanding data access patterns.
    - Some questions to assess data operations:
        - What is the get/insert/update/delete/invalidate operation ratio? i.e., 20 % inserts and/or updates, 80% gets, 3/2 get/put ratio, how many/often deletes/invalidates,
        - If using OQL-based queries: ad-hoc queries and/or continuous queries? How complex? How often? What would be the size of intermediate and final query resultsets, if you know them/can forecast?
        - Are you using messaging semantics in GemFire (e.g, RegExp or list key-based interest registration)? How often and how big is the set of keys that you are subscribing to?
        - Are you using function routing and execution service? How?

- **Anything else relevant to data access patterns: What peak loads** are you expecting? (i.e., beginning of the day, end of the day, random)

c) Throughput and latency requirements? What quality of service do you expect?

- These requirements might be different for different types of data and different for publishers, consumers, and, cache servers. Operational throughput and latency are very much related to the type, and concurrency of operations and are also intimately related to the available physical network bandwidth and latencies (e.g., data center backbone and MAN/WAN links bandwidth and latency).

d) Availability requirements specific to your project. For example:

- **HA data / redundancy level**: Do you require two (or more) copies of all/parts of your in memory data for HA?
- **Back up**: Do you require GemFire's persistence to disk and/or guaranteed lazy-write-behind to a RDBMS (i.e., Oracle, Sybase, SQL Server) or to another data sink such as a distributed file system?
- **Data loading** from data sources: Do you require data loading from a RDBMS, distributed file system, XML file repository, market data feed(s), other(s)? How often does data loading/refreshing

  occur (e.g. bulk loading at the beginning of the day or week plus continuous lower rate updates/ inserts)?
- **Network bandwidth**: What are the available network bandwidth and latencies available between GemFire cache nodes (i.e. 100Mbps, 1GbE, 8 or 10Gb Infiniband)?
    - Also, if connecting multiple Distributed Systems with each other, what is the bandwidth/latency between them and what would be the data operation rate expected between those Distributed Systems? For example, are you connecting geographically dispersed cache clusters via a WAN or MAN link (i.e. 200MB with 100ms average round-trip)?
- **Multiple caches in multiple geographies:** Is your configuration such that you need to transfer cache entries to other caches in different systems or geographies? Do you need to transfer a subset of GemFire regions or subset of data entries to another Distributed System via GemFire WAN Gateways?

Again the answers to these questions determine your next steps in the architecture solution design process. For example, if you need persistance to disk, how many regions are going to be written to disk? If multiple, disk IO might become a bottleneck in which case instead of using a local disk, you would want to use an array of local disks, or, alternatively write to shared drive.

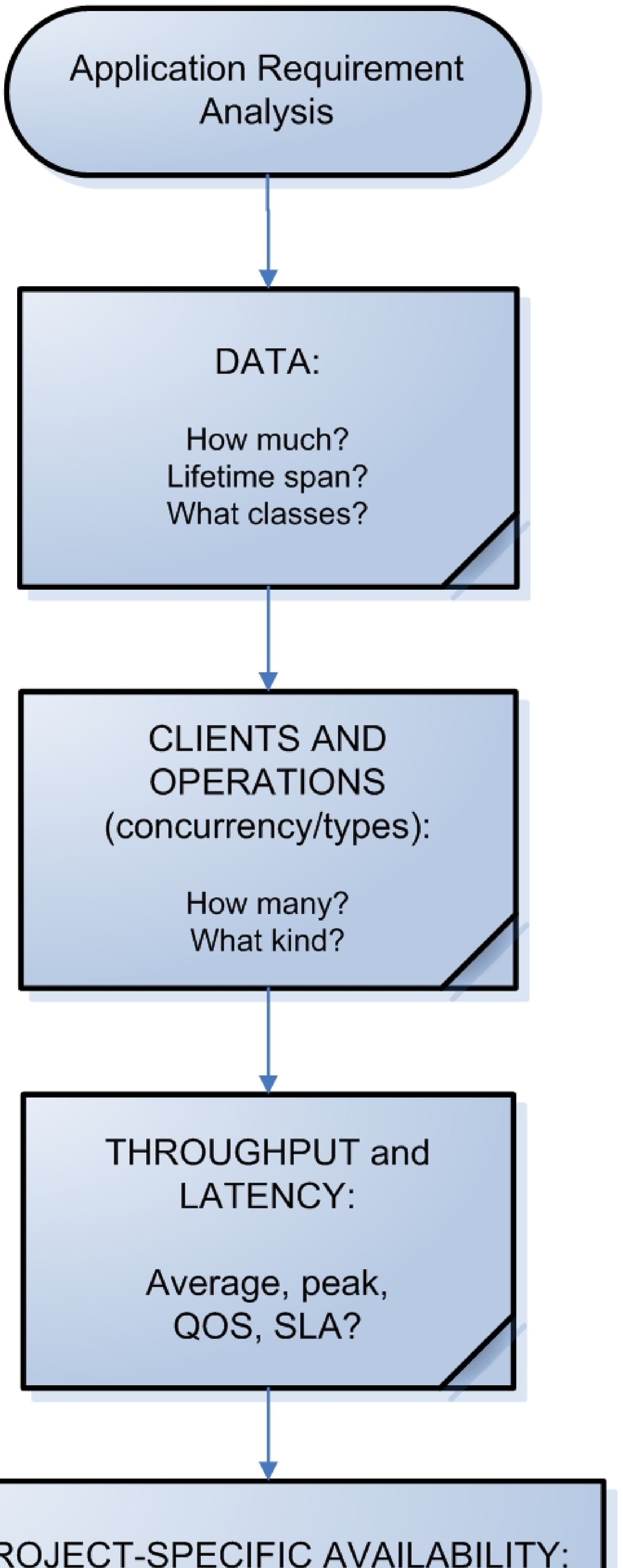

Application Requirement Analysis
DATA:
How much?
Lifetime span?
What classes?
CLIENTS AND OPERATIONS (concurrency/types):
How many?
What kind?
THROUGHPUT and LATENCY:
Average, peak, QOS, SLA?
PROJECT-SPECIFIC AVAILABILITY:

Figure 1. Application Requirement Analysis at a glance

2. Choose the appropriate GemFire data region design based on the above requirements (i.e. replicated data regions, partitioned data regions, or a mix/ hybrid region solution)

Core guidelines for GemFire data region design for 32-bit platforms:

- If you have a small data set (e.g. < 4GB) and a read-heavy requirement, you should be using replicated regions.
- If you have a small data set (e.g. < 4GB) but it is CPU-intensive or you need to execute parallel business logic on your data using GemFire's function execution service, you should be using partition regions.
- If you have a large data set and you are concern about scalability you should be using partition regions.
- If you have a large data set and can tolerate an on-disk subset of data, you should be using replicated regions with overflow to disk.
- If you have different data sets that meet the above conditions, then you might want to consider a hybrid solution mixing replicated and partition regions.

Please see section 'Making the right design choice for GemFire Data Regions' in this document for more detailed data region design guidelines.

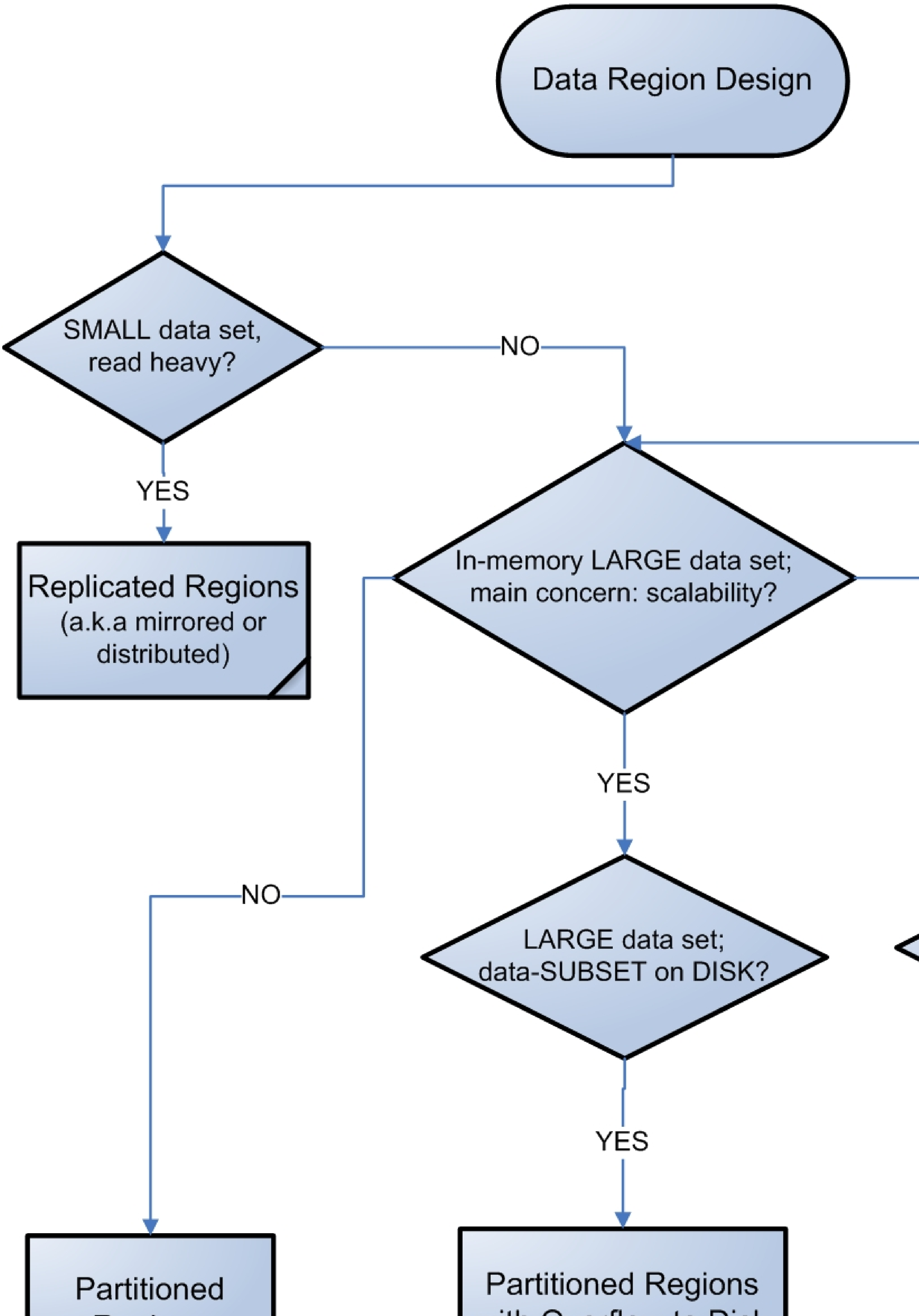

Data Region Design
SMALL data set, read heavy?
NO
YES
Replicated Regions (a.k.a mirrored or distributed)
In-memory LARGE data set; main concern: scalability?
YES
NO
LARGE data set; data-SUBSET on DISK?
YES
Partitioned
Partitioned Regions

Figure 2. Data Region Design Options for 32-bit and 64-bit platforms

3. Calculate the memory requirements for data

Here the solution architect will asses the entry overhead for each region.

a) Map your database data into objects or know your object sizes:

- Given your class structure, you would want to calculate the data object size (a.k.a. cache entry value or payload) and the region entry key size. Here we need to determine how many bytes do the "entry value" and the "entry key" take when they are serialized.
- In the simplistic scenario where the data mapping is one-to-one and the object structure consists of only primitive fields, the size of the data in GemFire memory (with no serialization) should roughly be the same.
- However GemFire typically stores/distributes each data region entry and its associated key in a GemFire serialized format which provides up to a 70% reduction in message size and network bandwidth savings (depending on the object graph).
- GemFire serialization reference information: To streamline serialization, GemFire provides the com.gemstone.gemfire.DataSerializable interface, which you can use in the place of vanilla Java serialization. The GemFire interface allows faster and more compact data serialization than the standard Java serialization. You can further speed serialization by registering the instantiator for your DataSerializable class through com.gemstone.gemfire.Instantiator. The instantiator knows how to instantiate the class, eliminating the need for reflection.

b) Calculate the size of cache entry for each object

For specific guidelines on the memory overhead introduced by the GemFire Enterprise Java API, see section 'Managing Memory' in the GemFire Enterprise Java System Administrator's guide http://www.gemstone.com/docs/6.0.0/product/docs Note that updated overhead values are listed in the paragraph and table below as a reference:

For each object added to a region, the GemFire Enterprise Java API consumes a certain amount of heap memory to store and manage the entry. This overhead is required even when an entry is overflowed or persisted to disk.

The per region entry overhead using a 32-bit JVM can be computed as listed below. For 64-bit JVMs, the number of bytes required for each object header and reference might vary with different JVM implementations and different JDK versions.

- GemFire introduces 67 bytes of overhead for each region entry. This value may vary because memory consumption for object headers and object references will vary for 64-bit VMs, different VM implementations, and different JDK versions.
- If you persist the region or overflow it to disk, add 30 bytes per entry.
- If statistics are enabled, add 16 bytes per entry. For details on enabling statistics in a region, see "statistics-enabled" topic in the "Region Attributes List" section (in the GemFire Enterprise Developer's Guide).
- When using the LRU (least recently used) eviction controller, add 16 bytes for each entry. For details about LRU eviction, see the "eviction-attributes" topic in the in the "Region Attributes List" section (in the GemFire Enterprise Developer's Guide).
- For each optional user attribute, add 52 bytes of VM memory (plus the space used by the user attribute object).
- For global regions, a distributed lock token may be needed for each entry. Each token uses 216 bytes of VM memory.
- For entry expiration, add 81 bytes of VM memory per entry
- For indexes used in querying, the overhead varies greatly depending on the type of data you are storing and the type of index you create.

The following table summarizes all region entry overhead for 32-bit and 64-bit platforms:

Configuration	32-bit platform	64-bit platform
Default	67	114
Default for VMs that the object was distributed to	83	179
Default, with statistics enabled	83	131
Default, but with idle entry or entry TTL expiration. Requires stats enabled.	174	289
Global region grantor and putter are same VM	233	413
Global region, putter VM only	166	279
Global region, grantor VM only	214	410
Default, but with region size LRU eviction with local invalidate action. Does not require statistics. No entries evicted	83	139
Default, but with region size LRU eviction with overflow to disk. No stats. All entries evicted.	115	195
Default, but with region size LRU eviction with overflow to disk and persistence. No stats. No entries evicted.	123	195
Default, but with a persistence and no op log rolling	107	171
Partitioned Region. 1 host. 113 and 499 buckets	83 + serialization overhead (91)	~174

Table 1. GemFire API - Region Entry and Index per Entry overhead

c) Add any other GemFire overhead (i.e., index overhead)

Again, please refer to the table above.

4. Calculate GemFire cache server memory requirements

a) Overhead associated with cache server to cache client TCP/IP connections

- For improved latency, GemFire provides two connection management options:
  - Single thread per cache client connection: Use a single thread to service each cache client connection. There is a thread stack overhead per connection (at a minimum 256KB to 512 KB; you can set it smaller to 128KB). With no checking of limits, a rogue application can easily overrun the stack resulting in "Out of Memory" conditions.
  - Multiplexing connections to a set number of cache server threads: multiplexing many connections to a configurable set of service threads (preferred option). Use the policy

configuration where the administrator can explicitly limit the number of concurrent connections to a cache server. Cache clients will try other available cache servers once the configured limit is reached.

b) Overhead associated with socket buffers (configurable):

- You can configure the buffer size of the sockets connecting a cache server to a cache client through the cache configuration file and through the Java API. The default setting is 32,768 bytes. You should make sure the size is at least as large as the largest key and value that you expect to send plus 100 bytes for overhead.
  Server side buffers: To configure the socket buffer size for sockets originating in the cache server, use either the 'socket-buffer-size' attribute in the 'cache.xml' or the 'CacheServer' method, 'setSocketBufferSize'.
  Client side buffers: To configure the socket buffer size on sockets originating in the client, add the 'socketBufferSize' parameter to the declarations or to the Java properties.

c) Overhead associated with GC tuning

- Garbage collection, while necessary, introduces latency into your system by consuming resources that would otherwise be available to your application. If you are experiencing unacceptably high latencies in application processing, you might be able to improve performance by modifying your JVM's garbage collection behavior. Garbage collection tuning options depend on the Java virtual machine you are using. For more on GC Tuning with GemFire, read GC Tuning

d) Fixed GemFire executable overhead

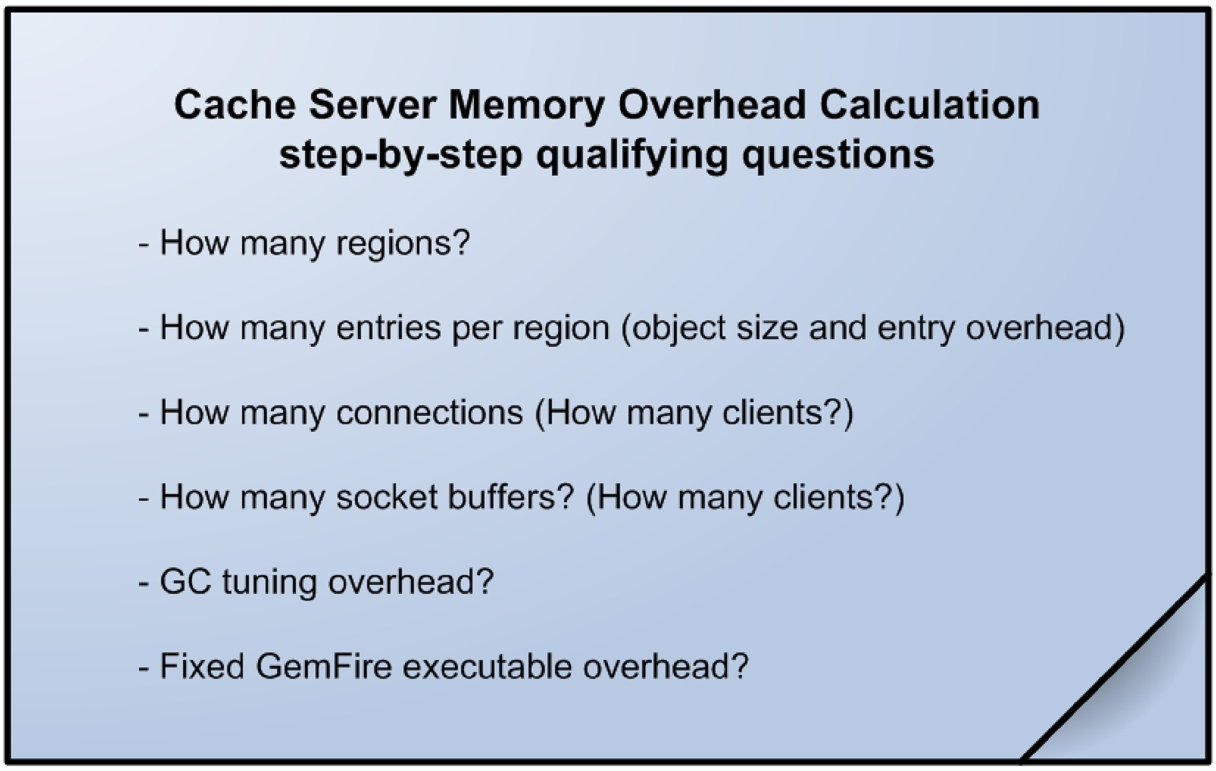
- In general it depends on the number of regions, number of region entries, and number of clients. Please see section 'Managing Memory' of GemFire Enterprise Java System Administrator's guide for more reference information.

5. Finally, with all the overhead calculated, figure out the number of nodes required to host the cache server farm.

Here we need to estimate the number of server nodes required to satisfy the data peak traffic and the high availability requirements.

The number of recommended clients per cache server will depend on a number of variables: concurrency and type of the operations, available network bandwidth, latency requirement, CPU utilization and number of available cores per cache server process, to name the most important. The most important thing here is to know the peak load characteristics of your applications on your GemFire-based data grid deployment.

For example, eight cache servers with 1.5GB heap space for data and 1,000 cache client based processes (data consumers and/or producers) per data center for a single grid application. The GemFire solution has a direct relation with the region type(s) (distributed, partition, distributed with overflow to disk).



Cache Server Memory Overhead Calculation step-by-step qualifying questions

- How many regions?
- How many entries per region (object size and entry overhead)
- How many connections (How many clients?)
- How many socket buffers? (How many clients?)
- GC tuning overhead?
- Fixed GemFire executable overhead?

Figure 3. Calculation of Cache Server Memory Requirements

Making the right design choice for GemFire Data Regions

Once the data, availability, and operation requirements are known, the next step would be to identify the right GemFire data region design that will be most suited for your application. Four main cache server region options are available as described below. Please note it is also recommended and possible to have a mix or hybrid configuration using regions of any of the four region type options below.

In parallel or right after assesing how many regions and what region types, the solution architect would also want to detemine how many cache server instances will be required to host your data, and, on how many nodes.

- Replicated data Regions (RR)
- Partitioned data Regions (PR)
- Replicated data Regions with disk-overflow
- Partitioned data Regions with disk-overflow

The number of cache server instances per node will depend on:

- How much total amount of data you need to store and how much available RAM memory per machine and how many machines
- 32-bit or 64-bit cache server JVMs
- How many total cache clients and what would be the client/server ratio
- What is the available aggregate server-to-client network bandwidth per machine and per cache server cluster

GemFire Region Options:

As GemFire Regions are stored in memory please remember that:

- On 32-bit platforms, the maximum memory that a single process can allocate ranges from approximately 1.5 GB to 3.5 GB.
  The upper bound of the process memory is roughly about 1.5 GB in a 32-bit architecture (i.e., Windows, Red Hat, SUSE). For Red Hat Enterprise ES/AS 4.0 the upper bound of the process memory is roughly about 2.7GB in a 32-bit architecture.
- Although a 64-bit platform increases the theoretical memory ceiling to approximately 16.8 million TB, today's 64-bit platforms can support only up to 128 GB RAM. For example, GemFire deployed on a 64 GB server yeilds 62 GB of memory allocated by a single process.

A. Replicated data Regions:

In this design, all entries in a data region are always locally managed in memory of a single process (a single cache server or cache client process). One or more replicated processes can also maintain the data in a consistent fashion. Replicated data regions will typically be the choice when the amount of data being managed in the data region can fit in the process memory.

Many replicated copies of data regions allow concurrent clients to load balance across many cache server nodes. A group of cache server nodes containing replicated copies of data regions is sometimes referred as a cache server replication group.

64-bit replicated cache servers should only be used when data access patterns are such that GC (Garbage Collection) time is minimized, for example, in an application that is almost exclusively reads, with very few updates. This configuration also works better with a small number of large objects than with a large number of small objects, since fewer objects takes less time to GC. The limiting factor with a 64-bit server configuration is garbage collection time. As heap size increases, the amount of time needed to do a GC grows. We have seen production environments running 30GB 64-bit Sun Microsystems JDK 1.5 Cache Server JVMs on RHEL successfully.

When the entire data set cannot fit into memory, applications can configure nodes to manage different data regions and continue to use replicated data regions. The idea here is to manually partition your data set into subsets that can still fit into replicated data regions.

As a rule of thumb, go with the replicated cache design, if the data set size is small and the application is read heavy.

B. Partitioned data Regions (PR):

A partitioned region can manage up to large volumes of data by partitioning that data into manageable chunks (buckets) and distributing it across multiple cache server instances running on multiple nodes or machines. In brief, here the data is automatically spread across multiple nodes: data is managed in buckets and buckets are spread across the GemFire managed distributed system cache server instances.

Partitioned regions can be configured with a redundancy level (i.e. two or more copies of the data set in memory) - this is to ensure that each bucket is consistently managed in at least one other node to protect for HA. Additionally partitioned regions can be configured to overflow and/or presist cache entries to disk providing further data scalability. In general it is recommended to have a redundancy level of no more than 1 (e.g., redundancy-level='1'), to minimize the memory footprint.

Partitioned data region cache design is the most suitable when your requirements call for:

- Large data sets: the amount of data to be managed is rather large (and it does not fit into one process space), and, the application wants to manage all the data in memory for high performance. Partitioning provides scalability and reliability for applications where the data set is very large (again, very large meaning data set does not fit into one process space).

- Scalability is a primary concern for the application: By partitioning the data across multiple nodes, many concurrent clients can capitalize on the CPU cycles available on these distributed nodes and data access is less constrained by the node level network bandwidth limitation (i.e., 1Gbit full duplex Ethernet).

- Update heavy environment: partition regions are more suitable than a replicated data region design in an update heavy environment.

- Easily add or remove nodes as the data volume requirements or the concurrent operations change.

It should be noted that Partitioned data Regions (PR) comes with the following tradeoffs compared with the replicated cache design:

- ◦ Any client access to a PR may require up to two network hops for data access compared with the always single network hop with replicated regions. This is due to the fact that client connections are typically configured to stick to a cache server and all client requests are always delegated to the connected cache server. With the data partitioned across all the cache servers, the cache server receiving the request (the delegator) may have to hop over to another cache server to satisfy the request.
- ◦ Not all the features such as transactions are available over partitioned data regions. Please consult the GemFire developer's guide for capabilities not available with partitioned regions in the current release.

C. Replication data regions with disk-overflow:

This design approach is similar to replicated data region design above and suitable when the volume of data is larger than what will fit in memory (i.e. >1.3GB on RedHat 3.0, >2.7GB on RedHat 4.0 or above). It should be noted that GemFire provides a "shared nothing" disk architecture - each member cache owns and operates on the disk files independent of each other. So, in a typical usage of disk overflow each data region can configured to write the data to one or more disk files on local disk. Each replicated member will concurrently do disk writes to its local disk. The data that should be managed in memory is configurable - it is either based on entry count or percentage of heap or amount of memory consumed. GemFire uses a LRU algorithm to manage the most frequently used data in memory at all times.

It should also be noted that GemFire only overflows data region values to disk. All data region keys are always managed in memory.

Using Disk regions is most suitable when:

- The quantity of data managed is large.Partitioned data management is not appropriate, either due to lack of available memory or application data access characteristics (only a small percentage of the data is active and will be in memory).
- Main memory replication is not sufficient for data availability. Data has to be stored on disk.

D. Partition data Regions with disk-overflow:

A partitioned region manages large volumes of in-memory data by automatically partitioning it into manageable chunks (buckets) and distributing it across multiple Cache Server JVMs running on multiple machines (a.k.a. nodes). The memory in each member that is reserved for the partitioned region's use is called a partition of that region.

This design approach is similar to partition data region design above and suitable when the volume of data is larger than what will fit in memory in a partition region configuration. It should be noted that GemFire provides a "shared nothing" disk architecture - each member cache owns and operates on the disk files independent of each other. So, in a typical usage of disk overflow each partition data region can be configured to write the data to one or more disk files on local disk (or shared drive). Each cache member will concurrently do disk writes to its disk. The data that should be managed in memory on each partition is configurable - it is either based on entry count or percentage of heap or amount of memory consumed. GemFire uses a LRU algorithm to manage the most frequently used data in memory at all times.

It should also be noted that GemFire only overflows partition data region values to disk. All partition data region keys are always managed in memory.

Using partition disk regions is most suitable when:

- The quantity of data managed is larger than what would fit in an in-memory partitioned region.
- Have only in-memory large replicated regions, or, partitioned regions w/o overflow is not appropriate, either due to lack of available memory or application data access characteristics (only a percentage of the data is active and will be in memory).
- Main memory region storage is not sufficient for data availability. Data has to be stored on disk.

Resource Planning:

A. Estimating the resource requirements with a Replicated Region design

- Estimate the number of redundant copies of data you need (e.g., 1 or 2).
- Calculate the total size of data object.
- Calculate the size of data entries in GemFire.
- GemFire internal data structures overhead.
-
    - **Transport buffers:** For UDP (User Datagram Protocol) communications, the size of the buffer being used is configurable. See GemFire Enterprise Developer's and System Administrator's guides for more details on UDP buffer sizing. UDP can be used for both unicast and multicast messaging. You can configure UDP multicast for messaging on a per-region basis by setting the multicast-enabled attribute to true.
-
    - **Client Connection memory overhead:** GemFire servers use a single thread to manage each incoming client connection. Each thread has a stack overhead of either 256KB (32-bit architecture) or 512KB (64-bit architecture).
-
    - **Peer connection memory overhead:** Each cache server node manages two p2p (peer to peer) connections to every other peer member (cache server) in the distributed system. There is a thread that serves each peer connection and comes with the thread stack overhead.
-
    - **Other overhead:** There is additional overhead associated with GemFire classes being loaded, JVM overhead, region metadata, etc. This represents the baseline memory requirements for GemFire and is no more than 50MB. When the number of data regions being managed becomes unusually large then this overhead can be larger.
-
    - **Garbage collection overhead:** There is not an easy calculation here. It depends on the amount of garbage created at any given time, the latency expectations from the applications (i.e., long lived data vs. frequently accessed data), and, the type and concurrency of cache operations (e.g., puts/inserts vs. deletes vs. ad-hoc and Continuous Queries). We recommend giving from 20% to 30% memory headroom for any garbage that builds up in the cache server JVM during peak conditions.

5) 32-bit vs. 64-bit architecture

32-bit architecture available memory per process is limited to about 1.7GB of total heap. As a rule of thumb the maximum recommended data size is 1.3 GB or less. Running more 32-bit cache servers vs. fewer amounts of 64-bit cache servers can typically help with reducing GC related pauses.

6) Estimating CPU requirements

This totally depends on the number and type of concurrent activity (i.e., reads, inserts, updates), the data sizes, and performance requirements. As a rule of thumb every cache server should have at least 1 core or CPU. Cache servers are multi threaded processes and would benefit from running on more than 1 core or CPU.

It is advisable that the CPU utilization be on average no greater than 80%; you would need to add a new cache server machine (or more core/CPUs) if the aggregate CPU utilization on the server box is greater than 80%. Please note this is not a strict number but a reference number (as it depends on hardware specs, complexity and concurrency of operations, server instance head count, etc).

The GemFire statistics engine captures the CPU utilization statistic periodically in a statistics database. The GemFire Visual Statistics Display tool (a.k.a., VSD) allows administrators to monitor the utilization in real-time for one or all cache servers in the cache server cluster. Applications also have the option to use the programmatic Admin API (or JMX) to introspect the statistics and proactively start an additional cache server when the utilization exceeds a certain set threshold. It is possible to run multiple GemFire Cache Server instances on a multi-core/multi-cpu machine, as long as the CPU usage average stays below 80%. Please note again this is not a strict number but a reference number. In any case, the number of Cache Server instances should never exceed the number of available cores, however.

7) Estimating bandwidth requirements:

Your network backbone bandwidth is typically the most limiting factor in the performance of a GemFire solution. Typically GemFire will 'perform' as fast as the available network bandwidth allows (i.e., when doing fire hose get operations the upper bounds are strictly the backbone bandwidth on the cache server replication group).

When using the replicated region design, when a cache client performs a new operation (i.e., insert, update, invalidate, delete), the client 'sends' the operation request to the cache server it is actively connected to (the cache server is just one hop away from the client).Then that cache server will propagate the operation information to all the other cache servers that replicate that region. Thus the network bandwidth utilized when using mirrored (a.k.a. replicated or distributed) regions is higher that when using partition regions.

Also as more machines are added to the cache server replication group, the use of network bandwidth increments significantly as any operation needs to be propagated to all other cache server machines (in the cache server replication group).

B. Estimating the resource requirements with a PR (Partitioned Region) design

1) Estimate the number of redundant copies of data you need

Typically, you only need one extra copy of data being managed for high availability. But, sometimes, more redundant copies of data could be configured for smaller data regions or when the concurrent query load on the data region is very high.

2) Calculate the total size of data object

First map your database data to objects - in the simplistic scenario where the data mapping is one-one and the object structure consists of only primitive fields, the size of the data in GemFire memory should roughly be the same. In GemFire all region entries have an associated key, so, you would need to calculate the object size corresponding to the key as well as the value.

3) Calculate the size of data entry in GemFire

GemFire Cache servers manage data objects in a serialized form - either using Java serialization or GemFire native data serialization. The use of GemFire serialization can result in more compaction.

GemFire provides the interface, 'com.gemstone.gemfire.DataSerializable' that you can use in place of Java's standard object serialization. The interface allows faster and more compact data serialization than the standard serialization, typically compacting a payload up to 70%. You can further speed things by registering your 'DataSerializable' instance with the data serialization framework, eliminating the need for reflection to determine the proper serializer to use.

4) GemFire internal data structures overhead

- Transport buffers:* For UDP (User Datagram Protocol) communications, the size of the buffer being used is configurable. See GemFire Enterprise Developer's and System Administrator's guides for more details on UDP buffer sizing. UDP can be used for both unicast and multicast messaging. You can configure UDP multicast for messaging on a per-region basis by setting the multicast-enabled attribute to true.

When using TCP communications and a release prior to GemFire 5.0.1, the number of buffers is calculated as follows:

Number of buffers = 2 x (Number of concurrent p2p connections from other servers + number of concurrent client connections to the server)

The size of each buffer is the maximum serialized message size sent or received from a client or peer. So, for instance, if the objects being managed have a maximum size of 1MB, each buffer will grow to 1MB. The default size of each buffer is 128KB.

- Client Connection memory overhead:* GemFire servers use a single thread to manage each incoming client connection. Each thread has a stack overhead of either 256KB (32-bit architecture) or 512KB (64-bit architecture).

- Peer connection memory overhead:* Each cache server node manages two p2p (peer to peer) connections to every other peer member (cache server) in the distributed system. There is a thread that serves each peer connection and comes with the thread stack overhead.
- Other overhead:* There is additional overhead associated with GemFire classes being loaded, JVM overhead, region metadata, etc. This represents the baseline memory requirements for GemFire and is no more than 50MB. When the number of data regions being managed becomes unusually large then this overhead can be larger.
- Garbage collection overhead:* There is not an easy calculation here. It depends on the amount of garbage created at any given time and the latency expectations from the applications (i.e., long lived data vs. frequently accessed data). We recommend giving from 20% to 30% memory headroom for any garbage that builds up in the server JVM during peak conditions.
- 32-bit vs. 64-bit architecture:* 32-bit architecture available memory per process is limited to about 1.7GB of total heap. As a rule of thumb the maximum recommended data size is 1.3 GB or less. Running more 32-bit JVM cache servers vs fewer amount of 64-bit servers can typically help with reducing GC related pauses.

5) Estimating CPU requirements

This totally depends on the number and type of concurrent activity (i.e., reads, inserts, updates), the data sizes, and performance requirements. As a rule of thumb every cache server should have at least 1 core or CPU. Cache servers are multi threaded processes and would benefit from running on more than 1 core or CPU.

It is advisable that the CPU utilization be on average no greater than 80%; you would need to add a new cache server machine (or more core/CPUs) if the aggregate CPU utilization on the server box is greater than 80%. Please note this 80% ceiling is not a strict number but a reference number (as it depends on hardware specs, complexity and concurrency of operations, server instance head count, etc).

The GemFire statistics engine captures the CPU utilization statistic periodically in a statistics database. The GemFire Visual Statistics Display allows administrators to monitor the utilization in real-time for one or all servers in the replicated group. Applications also have the option to use the programmatic Admin API (or JMX) to introspect the statistics and proactively start an additional server when the utilization exceeds a certain set threshold. It is possible to run multiple GemFire Cache Server instances on a multi-cpu machine, as long as the cpu usage average stays below 80%. The number of Cache Server instances should never exceed the number of available cpus, however.

6) Estimating bandwidth requirements:

When using the partition region design, when a cache client performs a new operation (i.e., insert, update, delete), the client 'sends' the operation request to the one cache server it is actively connected to. If the redundancy level is set to one (two copies of a data bucket/entry on the network), then the 'cache client' operation would have to be propagated to the second machine containing the entry for that replicated region. This would be, at best, a one hop operation and, at worst, a two hop operation. This cost does not increase as new cache server machines are added to the replication group. Thus there is less overall network bandwidth overhead when using partitioned regions than when using replicated regions. Also when using partitioned regions the network bandwidth utilization increments linearly as more machines are added to the cache server partition region group.

C. Estimating the resource requirements with a RR (Replicated Region) with GemFire's overflow-to-disk design

1. Same process as for replicated region requirements:

You would follow an almost identical process than when using replicated regions with no overflow-to-disk. Only you have to add the additional memory count resulting from using overflow-to-disk and LRU eviction controller.

2. Regions with overflow to disk requirements:

- If you persist the region or overflow it to disk, add 40 bytes per entry.

- When using the LRU (least recently used) eviction controller to overflow-to-disk, add 16 bytes for each region entry. For details about LRU, see the GemFire Enterprise Developer's Guide.

D. Estimating the resource requirements with a PR (Partitioned Region) with GemFire's overflow-to-disk design

1. Same process as for partitioned region requirements:
You would follow an almost identical process than when using partitioned regions with no overflow-to-disk. Only you have to add the additional memory count resulting from using overflow-to-disk and LRU eviction controller.

2. Regions with overflow to disk requirements:
- If you persist the region or overflow it to disk, add 40 bytes per entry.

- When using the LRU (least recently used) eviction controller to overflow-to-disk, add 16 bytes for each region entry. For details about LRU, see the GemFire Enterprise Developer's Guide.

Example 1: 1,000 cache client grid application, 10GB active data stored in cache server cluster

Example Requirements Summary:

GemFire Enterprise data grid solution is used to manage huge amounts of data in computationally and data intensive problems. We provide here a GemFire-based grid computing application for one data center. The GemFire Enterprise Java product (Java API) and its C+/C# native client (C/C# APIs) are used as the 'highly-available in-memory data grid repository' providing in-memory data storing and messaging capabilities used by risk management, derivatives pricing, and algorithmic trading portfolio optimization Java/C+ grid applications.

Application requirements:

1. Total data size in cache: 10GB (cache server side)

This is the amount of data that needs to be stored in-memory in the cache servers. Data scalability is the main concern here.
10 GBytes = 1,024B x 1,000,000 objects total in memory (objects could be Java objects or just XML files stored as Java 'StringBuffer-s'

2. Maximum active data size on client: 1GB (cache client process side)

This is the maximum amount of data that needs to be in-memory on each of the clients. This client data would be a subset of the 10GB stored in the server side (and not necessarily the same subset on each client)
1GByte = 10KB x 100 objects total in memory

3. 1,000 processes (in one application) using the cache in one data center

These means 1,000 GemFire cache clients based grid computing engine instances. There could be more than one engine running on the engine boxes, typically one engine per available core.

4. Frequency and type of operations:

Aggregate cache client operation targets:
- Read rate: 1GB/sec = 100,000 fetches/sec of 10K entries
(0.5s compute time; 50 data entries read per compute; 1,000 processes)
- Create rate: 100MB/sec = 10,000 creates/sec. Each entry 10KB in size.
(0.5s compute time; 5 data items created per compute; 1,000 processes)
- Update rate: 100MB/sec = 10K updates/sec. Each entry 10KB in size.

5. Backbone bandwidth: 10Gb Ethernet (that is TCP/IP, no Multicast/UDP permited)

One data center with a backbone of 10Gb each way (TCP/IP supported). Because of the overhead introduced by the TCP/IP protocols, the available speed for GemFire is about 70% of that 10Gb. Need to use only TCP/IP communications. Reliable Multicast and UDP not allowed for client-server communications.

6. Disk utilization: none

To simplify example no Cache Server persistence/overflow to disk are required.

7. Node/Host specifications:

Client hosts:
CPU: 2-core
OS: 32-bit OS (either Windows or Linux RH ES/AS)
JVM: 32-bit Sun Microsystems JVM (JDK 1.4 or JDK 1.5)
RAM: 8GB
Server hosts:
CPU: 4-core (i.e. 2-core 2 CPU)
OS: 32-bit OS with 'huge mem' turned on (or it could be HRES v3.0) which means 4GB of RAM memory is available per process.
JVM: 32-bit Sun Microsystems JVM (JDK 1.4 or JDK 1.5)
RAM: 16GB

8. Grid scheduling software:

Any of the following or others would very smoothly work/integrate with GemFire: DataSynapse GridServer/FabricServer, or Platform Symphony/EGO, or CONDOR.
Please note that it is very straight forward to have a computing grid engine or a client application (e.g., a trader workstation app) use the GemFire cache client API to access the data stored in the Cache Server cluster. Client load balancing is automatically taken care for you by GemFire. HA and DR capabilities are also automaticaly taken care for you.

GemFire-based solution analysis:

Based on the data, client, operation, and throughput requirements we can start taking the following design steps:

- Choose type of regions: As you have 10GB worth of 10KB data payloads need to be stored in memory in a GemFire cache server cluster (a.k.a., Distributed System) with no overflow/persistence to disk. We would recommend a partition region design that can scale to accommodate larger data sizes and larger number of clients overtime.
- When using Linux RedHat ES/AS 3 to host GemFire cache servers, we will have about 1.5 to 1.8GB of heap space per cache server for data. Taking the most pessimistic approach, 1.5GB heap/server, we would need 7 cache server JVMs to store 10GB of data using the partition region approach.
- As we are using 4-core machines to host our cache server cluster cloud, we could distribute our 7 cache servers on a 2-per-machine basis. We will use a total of 8 cache servers to round up our calculation with the benefit of having some extra latency/performance enhancement, and, better HA if one machine or cache server would fail. We will run 2 cache servers per cache server box; for a total of 4 cache server boxes.
- The aggregate cache client to cache server network bandwidth and latency already meets our operation concurrency requirements.
- Also, 8 Cache Server cluster will load balance the 1,000 concurrent cache client based grid engines. That is about 125 cache clients per cache server, which is a very comfortable client/server ratio.

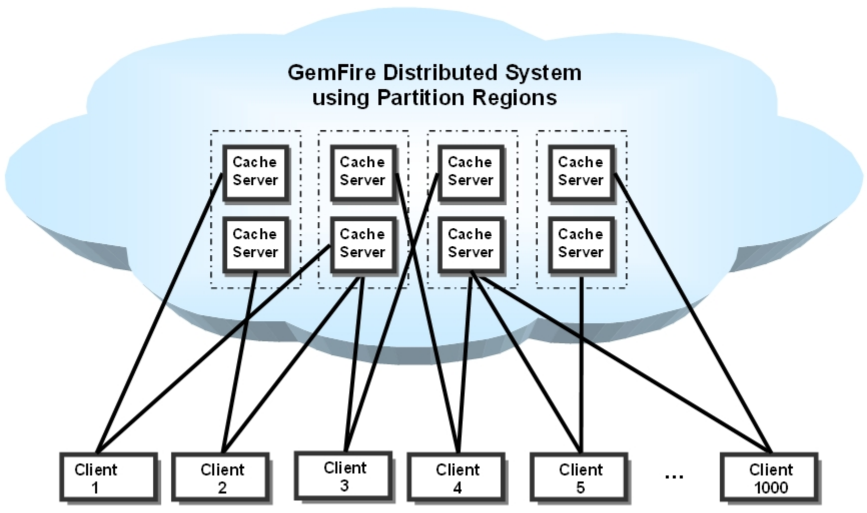


Figure 4. Client-Server topology based Solution - using Partition Regions

References:

These recommended articles can help you to better understand the process of GemFire adoption, and, data resiliency considerations, and GC tuning to meet your project requirements:

1. Migrating Applications to GemFire
2. Data Resiliency Considerations: http://www.gemstone.com/docs/html/gemfire/6.0.0/DevelopersGuide/DataConservation.12.19.html
3. Garbage Collection Tuning