Evaluating the Performance of Data Caching Frameworks
GigaSpaces XAP versus VMware vFabric GemFire
# Table of Contents

Executive Summary ................................................................................................................. 1  
  Key Results............................................................................................................................... 2  
Business Context....................................................................................................................... 4  
Data Challenges ....................................................................................................................... 5  
Solution Overview: Distributed Data Caching ........................................................................... 5  
Choosing a Data Caching Solution ............................................................................................ 6  
GigaSpaces XAP versus VMware vFabric GemFire ................................................................. 8  
  Benchmark Details.................................................................................................................. 8  
  Benchmark Environment ..................................................................................................... 10  
  Remote Write/Read Benchmark ............................................................................................ 11  
  P2P/Embedded Write/Read Benchmark .............................................................................. 12  
  Index Tests............................................................................................................................ 14  
  Batch Write/Read Benchmark ............................................................................................. 16  
  Local Cache ......................................................................................................................... 18  
Meeting the Needs of the Enterprise ....................................................................................... 19  
Conclusions............................................................................................................................. 20  
Appendix – Performance Factors............................................................................................ 21  
  Comparison of System Defaults .......................................................................................... 21  
  Functional Comparison....................................................................................................... 23  
    Cache Regions .................................................................................................................. 23  
    The Space – Natural In-Memory Object Store ................................................................. 24  
    Replication vs. Partitioning ............................................................................................. 25  
    Classpath and Data Versioning ....................................................................................... 25  
    Rebalancing – Does the System Maintain Consistency upon Failure? ............................ 26  
    Partitioned Regions with Redundancy ............................................................................ 26  
    Capacity Planning ............................................................................................................ 26  
    Cache Configuration ....................................................................................................... 28  
    System Recovery from Failure ....................................................................................... 28  
Disclaimer............................................................................................................................... 29
### Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Benchmark Basic Architecture</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Remote <em>Write</em> Benchmark, Regular Payload Results</td>
<td>11</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Remote <em>Read</em> Benchmark – Regular Payload Results</td>
<td>11</td>
</tr>
<tr>
<td>Figure 4</td>
<td>P2P/Embedded Benchmark Architecture</td>
<td>12</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Space-Based-Architecture</td>
<td>12</td>
</tr>
<tr>
<td>Figure 6</td>
<td>P2P/Embedded Write Benchmark – Regular Payload results</td>
<td>13</td>
</tr>
<tr>
<td>Figure 7</td>
<td>P2P/Embedded Read Benchmark – Regular Payload results</td>
<td>13</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Remote Write Benchmark – Indexed vs. Non-Indexed Payload Results</td>
<td>14</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Remote Read Benchmark – Indexed Payload Results</td>
<td>15</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Remote Batch Write Benchmark – Regular Payload Results</td>
<td>16</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Remote Batch Read Benchmark – Regular Payload Results</td>
<td>16</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Remote Batch Write – Indexed Payload Benchmark Results</td>
<td>17</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Remote Batch Read – Indexed Payload Benchmark Results</td>
<td>17</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Local Cache Architecture</td>
<td>18</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Local Cache Benchmark Results</td>
<td>18</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Remote Write No-Ack Mode Benchmark Results</td>
<td>22</td>
</tr>
</tbody>
</table>
Executive Summary

Enterprise organizations not only depend on information technology in order to survive, they rely on it in order to compete and win in the global market. Since much of the value and differentiation within IT comes from applications, enterprise IT organizations must use applications that deliver the highest levels of performance, scale, availability, reliability and security.

However, today’s applications are more complex than ever. To meet enterprise requirements, developers from IT organizations, independent software vendors (ISV), software-as-a-service (SaaS) providers, web businesses or other organizations – use a variety of tools and frameworks. Data caching frameworks are particularly helpful in overcoming challenges at the data tier. They help overcome performance and scale limitations due to scale-up architectures, availability and reliability issues caused by single points of failure, and data loss and recovery issues from synchronization problems.

Just like applications themselves, individual data caching frameworks may have a number of functional differences including supported languages, ability to dynamically scale, event handling, monitoring capabilities and administrative control. Interestingly, even though improving the performance of applications is often considered the most important capability in a data caching framework, the performance attributes between frameworks varies significantly.

Since performance is so important, in-depth performance testing and analysis was done in order to compare differences between two popular frameworks. For this project, performance tests were run using VMware vFabric GemFire 6.6.1 and GigaSpaces XAP 9.0 Elastic Caching. Performance levels for data caching under various application scenarios was completed and documented in this report.

The tests showed that in the simplest scenario (client and cache running on the same machine) GemFire and XAP performed on a comparable level. However, in other benchmark scenarios that are more true-to-life – running the tests in a distributed environment, with indexing, batch operations, and more – GemFire performance dropped off sharply whereas XAP scaled smoothly with no drop in performance.

“Data caching frameworks are particularly helpful in overcoming challenges at the data tier.”
While the full details of the performance testing and analysis are found throughout the main body and appendix of this report, the key results from performance testing are listed below in both bullet and graphical form.

Key Results

- XAP is **57 times faster** than GemFire with batch write and **8 times faster** with batch read with an indexed payload.
- XAP is **7 times faster** than GemFire with batch write with a regular payload.
- XAP is **6 times faster** than GemFire with remote single read, and **3 times faster** with remote read with an Indexed payload.
- XAP is **3 times faster** than GemFire with single write in P2P (embedded) mode with a regular payload.
- XAP is **2.5 times faster** than GemFire with read from a local cache.
- XAP is **2.2 times faster** than GemFire with no-ack remote write.
Graphical Summary of Key Results

Remote - Single, Regular payload

Remote - Single, Indexed payload

Remote - Batch, Regular payload

Remote - Batch, Indexed payload

P2P - Regular payload

Local Cache Read
Business Context

During 2012, enterprise organizations throughout the world are expected to spend a combined total of over $2.5T (trillion) on information technology. Every business function including R&D, manufacturing, distribution and sales depends on IT; every business process including billing, payroll and purchasing depends on IT; and every business partner including suppliers, distributors and retailers depends on IT. Yet having an IT organization is not enough.

In today’s global economy, enterprise organizations must have industry leading IT in order to compete. When they don’t, the consequences often include low market share, slow growth and/or minimal profits. At a very basic level, industry leading IT means having better applications than your competitors – applications which produce better results and that are fast, safe and available when needed.

At the same time, today’s applications are more complex than ever. Developers build applications on virtualized and cloud infrastructures for greater flexibility and increased utilization. They use scale-out architectures to meet growing capacity demands and ensure performance requirements are still met. They also use mechanisms like distributed caching to reduce IO barriers and speed processing at the data tier.

Of course achieving flexibility, efficiency, performance and scale is only part of the challenge. If an enterprise does not have accurate and up-to-date data, they risk failure. All kinds of applications – from e-commerce and financial trading, to analytics and business intelligence – absolutely must have fast, safe and accurate access to data. In order to meet all these objectives, developers must have the proper tools for building applications.

Fortunately there are a variety of frameworks available which not only make developers more productive, but help create more powerful applications that meet the rapidly increasing needs of the enterprise. Depending on the particular framework, it may provide distributed code execution, event and message handling, management and monitoring, high availability, elasticity or others. Since data-related challenges are so common in modern applications, there are frameworks which have been specifically designed to focus on them.

“Industry leading IT means having better applications than your competitors.”
Data Challenges

Modern applications use data sets that originate from real-time streams, video, mobile Internet, financial markets, scientific measurements and many other sources. However, handling this data is not easy. It must be stored, transmitted, processed and re-used by a variety of applications – ultimately impacting everything in the entire IT stack, including servers, storage, networking, operating systems, middleware and applications.

Hard drives are getting replaced by solid state drives; network pipes are getting fatter; servers are getting more CPUs and cores, as well as larger caches; and operating systems are addressing more RAM. Yet none of this really solves the fundamental data challenges. In fact, more powerful hardware is almost always just a temporary solution.

Application developers depend on far more elegant solutions to solve the most difficult underlying data challenges which include these:

- Performance and scale limitations due to scale-up architectures
- Availability and reliability issues caused by single points of failure
- Data loss and recovery issues from synchronization problems

Distributed data caching solutions are used to address all of these data-related issues. They can also help transform application architectures to support greater distribution and scale.

Solution Overview: Distributed Data Caching

To overcome the toughest data challenges and manage the increasing volume, velocity and variety of data, modern applications often used distributed data caching solutions. These caching solutions may incorporate several or all of the following techniques:

- Parallel processing and scale-out architectures
- Staging data records closer to the servers that will process them
- Placing data in fast memory to avoid slower disk-based reads and writes
- Replicating data to improve both availability and reliability
Beyond these high-level techniques, distributed data caching solutions have other technical attributes to consider. Since applications are written in a variety of languages, it is important for any third-party caching solution to support the required languages and language frameworks. These may include Spring, Java, JPA, .Net, C++ or others. Similarly, a viable caching solution must support must support the right target environments which may include the traditional data center, public/private clouds, and/or hybrid clouds.

Caching solutions also differ in the types of management and administrative support they provide. Initial provisioning of cache resources and automated adjustments to maintain optimal utilization levels both go a long way toward making a caching solution easy to manage. Flexible management of topologies, replication, memory and partitions are also important capabilities. When it comes to performance, linear scalability is a required attribute.

Choosing a Data Caching Solution

Data caching solutions are clearly not all equal and should be chosen, in part, based on the particular needs of the application. Enterprises should also consider each of the following characteristics.

- **Performance:** This is often the most fundamental decision criteria since many enterprise applications are performance-critical and latency sensitive.

- **Scalability:** Loads fluctuate and demand changes, so solutions should scale up and down automatically and at minimal cost.

- **Interoperability:** Enterprise applications run in traditional data centers, public, private and hybrid clouds. They also depend on a variety of languages and platforms.

- **High Availability:** No downtime is acceptable for mission-critical systems and production applications need automatic recovery and failover mechanisms built in.

- **Full Control:** Enterprises require full control and system monitoring, whether through compatibility with systems already on site, or built into the caching solution.

- **Security:** Enterprises require enterprise-grade, transactional security for both admin functions and data, especially when working in a virtualized or multi-tenant environment.

“Caching solutions also differ in the types of management and administrative support they provide.”
Much can be learned about these attributes by following a typical software selection process. For example, product data sheets, demos, proof of concept (POC) projects, and limited implementations all help reveal the true capabilities of any data caching framework. However, evaluating highly technical characteristics such as performance often demands even more in-depth testing and analysis.

Even though application performance improvement is often considered the most important capability in a data caching framework, the performance attributes often vary significantly between frameworks. This makes performance analysis an even more important part of the selection process. After all, when enterprise applications fail to meet business requirements, the consequences often include low market share, slow growth and/or minimal profits.

To help uncover some of the differences in performance, in-depth performance testing and analysis was done with two popular data caching frameworks. For this project, performance tests were run using VMware vFabric GemFire 661 and GigaSpaces XAP 9.0 Elastic Caching. Performance levels for data caching under various application scenarios were captured and then documented in this report.

Please continue reading for in-depth information on the test benchmarks, process, environment, results and analysis.
GigaSpaces XAP versus VMware vFabric GemFire

Benchmark Details

Open Source Benchmark – Test It Yourself

All tests in this study were conducted using a benchmark based on the Yahoo! Cloud Serving Benchmark. You can download the entire benchmark code from github. The code includes both the GemFire and the GigaSpaces clients, so you can run the comparison between the two products yourself. The original Yahoo Benchmark has been revamped to support the complex scenarios we tested, such as queries and batch operations, and to provide more precise results when running extreme low-latency tests.

The GigaSpaces Client implementation and the GemFire Client implementation support the exact options tested, such as single and batch operations, remote client or P2P client, and regular and indexed payload.

What the Benchmark Service Tests

The Yahoo! Benchmark enables throughput tests (number of operations a client can perform) and latency tests with a constant throughput target. In both scenarios, throughput and latency are measured as a function of the concurrent activity. In our benchmarks we increased the number of concurrent threads performing the operations. In essence, these two numbers (throughput and latency) measure the products’ performance. Our tests focused on throughput to keep the comparison simple.

Test Focus

In terms of the operations tested, the tests focused on write and read operations. Other operations supported by GemFire and XAP, such as data removal and business logic execution, scalability across multiple nodes and high-availability tests, will be tested in the future. Our focus was on the most basic operations any system would need to perform to help you decide which cache products to short-list for evaluation.
**Regular and Indexed Payload**

The benchmark tested two types of data structures: **Regular payload** – a simple object with key and payload fields, with a payload size of 1K (the value object had a hashmap with a single entry, where its key is a string and the value is a byte array). And **indexed payload** – four fields, all indexed. In both modes, the key was generated by the benchmark application, constructed of the prefix “user” + a running number.

The regular object mode represents a key/value data retrieval scenario, where the application fetches a single entry from the cache, based on the entry key. In the indexed object mode the application fetches data using a SQL query or a template, based on one or more field values. In this case the application fetches multiple matching objects (batch read).

In the write operations tests, the benchmark client application loaded data into a cache node running on a different machine, ensuring the data is serialized and transported over the network. Read operations were tested in two modes: one with client and cache node running on different machines, where the client never read the same key more than once, and the second, where the client read the same key multiple times (testing local cache). Another test performed batch write and batch read operations (data query based on value attribute value). The last tested P2P architecture, where the client and cache ran within the same JVM.

**Testing matrix summary:**

<table>
<thead>
<tr>
<th>Test Scenario</th>
<th>Operation</th>
<th>Write</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single, Regular, Remote</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Single, Indexed, Remote</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Batch, Regular, Remote</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Batch, Indexed, Remote</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Single, Regular, P2P</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Single, Regular, Local</td>
<td></td>
<td>X</td>
<td>✓</td>
</tr>
</tbody>
</table>
Benchmark Environment

The benchmark was conducted using the following:

- **Client machine**: Sun JVM 1.7 u3, CentOS release 5.3 2.6.18-128.2.1.el5, Sun Fire X4600, 8 cores, 16 GB RAM.

- **Server machine**: Sun JVM 1.7 u3, CentOS release 6.2, 2.6.32-220.4.2.el6.x86_64, CISCO UCS 16 cores, 384 GB RAM.

- **Network**: 1 GB interconnect.

- **OS**: No OS virtualization. Bare metal OS.

- Single cache node with 15 GB heap size. No special JVM tuning. Client with 1 GB heap size.

- No special optimization was done when testing the products at the OS, JVM, or network layers. Factory defaults were used.
Remote Write/Read Benchmark

The basic test: remote client writing and reading a regular payload data structure from the cache.

![Remote Write – Regular Payload](image)

**Figure 2 – Remote Write Benchmark, Regular Payload Results**

![Remote Read - Regular Payload](image)

**Figure 3 – Remote Read Benchmark – Regular Payload Results**

**Results**

GigaSpaces performed better than GemFire on the Write test, reaching up to 22,000 operations per second with 8 threads (100% better than GemFire). On the Read test, both products performed similarly, reaching 30,000 operations per second with 8 client threads.

**Test Notes**

1. The local cache was disabled in both products. Each key was read back to the client application once. The number of keys was increased in same proportion as the number of client threads to maintain the same test duration in all tests.

2. We tested GemFire in Global mode because other options could result in inconsistencies. See the Data-Distribution/Network protocol – Early-Ack section for more details.
P2P/Embedded Write/Read Benchmark

The peer-to-peer (P2P) mode, also called embedded or co-located mode, is a popular deployment topology used when the application business logic and the cache/space instance require fast data access. There is no serialization or network overhead when the business logic accesses the data.

P2P architecture is the foundation for the GigaSpaces Space-Based-Architecture, in which all/some of the application components and the data grid are partitioned across the network, and each partition includes a data-grid instance and the relevant business logic. Remote clients can access the data grid as well, to push data or retrieve data. Any heavy-lifting business logic, such as data processing, is deployed with the data grid nodes, and consumes data transparently using the appropriate event container.

The SBA architecture cannot be implemented with GemFire because it does not have a transparent co-location proxy and has a different partitioning approach.
Here are the results for the Embedded/P2P mode:

**Results**

**P2P/Embedded Write benchmark:** GigaSpaces out-performed GemFire, reaching 160,000 writes per second, compared to GemFire, which was unable to surpass 55,000 writes per second with 4 threads.

**Read benchmark:** Both products performed similarly, reaching nearly 1.4 million reads per second with 4 threads.
Index Tests

Indexing is an essential requirement for every mission-critical system. Any system that uses a back-end store uses some sort of indexing mechanism to avoid full table scans.

![Remote Single Write Indexed vs. Non-Indexed Write Benchmark](image)

**Figure 8 – Remote Write Benchmark – Indexed vs. Non-Indexed Payload Results**

**Results**

In the write operations test, GigaSpaces reached almost 35,000 operations per second, which is 530% more than GemFire.

With even just a single index defined (in synchronous indexing mode), even with un-indexed data (no matching field to index within the value object) GemFire slowed down drastically on write operations, reducing performance by 84% when 8 concurrent threads write to the cache. Compare this to the impact on GigaSpaces write performance, which is close to zero impact.
Performance note: GigaSpaces comes with a very fast indexing engine designed for real-time applications looking to fetch objects based on content and not upon a single key. The indexing engine processes any newly created incoming object or updated/removed object, and updates the indexes list synchronously. The write operation is acknowledged only after the relevant index has been fully updated. This means any operation write/update/remove operation and its index update is atomic, i.e., queries executed immediately after an object update use an up-to-date index and fetch the correct result set. GigaSpaces supports placing an index on any field within the object, at any nesting level, including non-primitive fields such as hashtable keys.

Results
In the read operations test, using indexed field with 8 threads, GigaSpaces reached 30,000 operations per second, whereas GemFire performed only 12,000, a 240% difference.
Batch Write/Read Benchmark

Batch operations are critical for any application. Applications might need to load large amounts of data into the cache, or read large amounts of data from it. Reading large amounts of data is usually done through a query that also uses indexed data.

Regular Payload

![Batch Write Benchmark – Regular Payload](image)

**Figure 10 – Remote Batch Write Benchmark – Regular Payload Results**

![Batch Read Benchmark – Regular Payload](image)

**Figure 11 – Remote Batch Read Benchmark – Regular Payload Results**

**Results**

In the regular payload scenario, GigaSpaces reached 224,000 write operations per second, which was a 760% performance difference compared to GemFire. For batch read operations both products performed similarly, reaching ~100,000 reads per second with 8 threads. The batch size for the write and read tests was 100 objects. Our observations were that performing batch operations drastically impacted GemFire performance.
Indexed Payload

**Results**

The batch write benchmark for an indexed payload resulted in GigaSpaces performance of 281,000 operations per second, 5700% of the GemFire performance.

GemFire simply did not scale and did not perform with an indexed payload. GigaSpaces performance showed only negligible effect from indexing.

**Results**

In the batch read text on an indexed payload, with 8 threads, GigaSpaces reached 662,800 reads per second, out-performing GemFire by 883%. We observed that even with indexed data, in fetching a batch based on a simple query (single predicate query), GemFire failed to perform.
Local Cache

Local cache is critical component for many applications. In read-mostly scenarios, where a data item is repeatedly read by the application with very frequent changes, it is beneficial to have a local copy of it within the application memory address.

This is usually important for web applications that handle large amounts of concurrent connections. A slow and unsalable local cache can severely impact the entire application’s response time, creating a very poor user experience.

![Local Cache Architecture](image)

**Figure 14 – Local Cache Architecture**

**Results**

The local cache read test (regular payload) indicates that GemFire does not scale. GigaSpaces reached 8.2 million reads per second at 8 threads, a performance rate 250% that of GemFire, which stayed static between 3 and 8 threads.

This means that with GemFire, the more concurrent activity there is, the lower the effective performance of the application is in terms of operations per second. GigaSpaces, however, scaled consistently, no matter how many concurrent access activities took place.
Meeting the Needs of the Enterprise

Meeting the needs of enterprise applications requires even more than a great architecture and leading performance. A data caching framework must also address scale, interoperability, high availability and enterprise-grade reliability and security. Both GigaSpaces XAP and VMware vFabric GemFire demonstrate many of these strengths, as shown below. **Note:** The listed characteristics are not intended as a head-to-head comparison, but as an illustration of just some of the other selection criteria beyond performance.

*Some Enterprise Characteristics for VMware vFabric GemFire:*

**Scalability:** Dynamic data partitioning, distributed application functionality, automatic data rebalancing, and continuous client load-balancing drive increased scalability for applications.

**Interoperability:** GemFire runs on a Java Runtime Environment (JRE) in 32-bit and 64-bit mode on Windows, Linux and Solaris. Client nodes running C++, C# .Net and Java are supported.

**High Availability and Reliability:** Data replication across nodes and clusters for high availability – shared nothing parallel disk persistence provides cluster-wide high availability and failure handling.

**Data Center Ready:** Several management options are available, including a command-line utility, a Java Management Extensions (JMX) Agent, and GFMon, a graphical user interface monitoring tool.

**Security:** Security capabilities include cache server authentication, client authentication, SSL-based connection authentication, authorization of cache operations, and data modification based on authorization.

*Some Enterprise Characteristics for GigaSpaces XAP:*

**Scalability:** Smart load balancing and distributed elastic provisioning enable scaling to meet fluctuating loads in fast and demanding transactional applications.

**Interoperability:** Native support of popular languages (Java,.Net, C++), APIs (Object/SQL, Map/JCache, JMS, and more), and frameworks (Spring, JEE).
**High Availability and Reliability:** With an in-memory transactional engine, sub-second hot failover, self-healing, and disaster recovery capabilities across the WAN, XAP has no single point of failure and reduces unplanned downtime close to zero.

**Data Center Ready:** XAP provides a cluster-wide monitoring and management UI, as well as an administration API that provides programmatic control over its In-Memory Data Grid.

**Security:** XAP offers security for multi-tenant scenarios with multiple enterprise applications and provides SSL communication between all data grid components, authentication, authorization, and fine-grained security for specific operations.

**Conclusions**

To compete effectively, enterprises must use applications that deliver the highest levels of performance, scale, availability, reliability and security. However, this is not always easy, particularly when considering data-related challenges such as performance and scale issues, availability and reliability issues, and data loss and recovery issues.

Data caching frameworks are helpful in overcoming challenges at the data tier. They help overcome performance and scale limitations due to scale-up architectures, availability and reliability issues caused by single points of failure, and data loss and recovery issues from synchronization problems. Enterprises should consider all these and other attributes when selecting a data caching solution.

Even though improving the performance of applications is often considered the most important capability in a data caching framework, the performance attributes between frameworks varies significantly. The performance tests and analysis in this report showed that in the simplest scenario (client and cache running on the same machine) GemFire and XAP performed on a comparable level. However, in other benchmark scenarios that are more true-to-life – running the tests in a distributed environment, with indexing, batch operations, and more – XAP showed a number of performance advantages.
Appendix – Performance Factors

Comparison of System Defaults

GemFire comes with factory defaults that deliver reasonable performance for simple scenarios. The reason is simple – write activity is asynchronous, and read is done first from a local cache. It is an unreliable approach and can cause data inconsistency. For many mission-critical systems this behavior is unacceptable. Take execution of a stock trade as an example: It would be possible to get system acknowledgement that the order has been executed, but in reality it is not yet fully processed, replicated or persisted. In the event of a failure and an attempt to recover, trade data will be lost.

If the more reliable mode is selected – the **Global** mode, which involves synchronous write behavior, synchronous indexing, and turning off the local cache – the entire system performance drops dramatically.

From the [GemFire documentation](http://www.gemstone.com/docs/html/gemfire/6.0.0/DevelopersGuide/DataManagement.7.7.html):

> “With **global** scope, entries and regions are automatically locked across the distributed system during distribution operations (see [Locking in Global Regions](http://www.gemstone.com/docs/html/gemfire/6.0.0/DevelopersGuide/DataManagement.7.7.html)). All load, create, put, invalidate, and destroy operations on the region and its entries are performed with a distributed lock. This is the only scope that **absolutely guarantees consistency** across the distributed system. It is also the slowest. In addition to the implicit locking performed by distribution operations, regions with global scope and their contents can be explicitly locked through the application APIs. **This allows applications to perform atomic, multi-step operations on regions and region entries.**”

GigaSpaces XAP factory defaults are different. XAP provides total data consistency for both write and read. When performing write operations, acknowledgement is received from the server only after the data is fully committed, indexed, and replicated (if there is a backup) and persisted (if running in synchronous persistency mode). Data is read, by default, from the master copy and not from a local cache that might store stale data. GigaSpaces does support local cache and asynchronous write using the `com.gs.onewaywrite` and `com.gs.onewayupdate` properties.
Using these settings, GigaSpaces XAP outperforms GemFire:

See below *distributed-no-ack and distributed-ack behavior*:

From the [GemFire documentation](http://www.gemstone.com/docs/html/gemfire/6.0.0/DevelopersGuide/DataManagement.7.7.html):

**Choosing the Level of Distribution**

Each distributed region must have the same scope throughout the distributed system. Distributed scope is provided at three levels:

- **distributed-no-ack**—This setting dictates that distribution operations return without waiting for a response from other caches. This setting provides the best performance, but is also most prone to race conditions. A temporary disruption of the network transport layer, for example, could cause failure of a distribution to a cache on a remote machine.

- **distributed-ack**—This setting causes distribution to wait for acknowledgement from other caches before continuing. See also the related region attribute, early-ack. This is slower than distributed-no-ack, but covers simple communication problems such as the one described above. Race conditions are still possible, however, as there is no guarantee that updates sent from multiple caches arrive in the proper time order. Distribution operations may arrive out of order due to network latency and other differences between caches. For example, some caches may reside on the same machine while others may communicate over large networks. Because there is no locking, with either distributed-ack or distributed-no-acksce it is possible for two members to replace an entry value simultaneously.
Note: As mentioned above, the **Global** setting is the most accurate; however it is also the slowest. When choosing a product with which to run financial, healthcare, defense, or other mission-critical applications, both speed and consistency are critical factors.

### Functional Comparison

#### Cache Regions

Cache region roots are located within the OODBMS, enabling the storage area to be broken into logical buckets. GemFire cache regions are usually mapped to application *domain entities*. If an application has multiple domain model entities (such as Department, Employee), each entity must have its own region. This means the application needs to be programmed to use the correct region when reading and writing data from the GemFire cache.

Using the following job application data model as an example:
GemFire would require a separate Region for each class (Supplier, Contact, Position, School, etc.) to be able to index each entity without affecting other entities, or alternatively, have the entire model as one big graph where all objects are stored within the same region. This necessitates fetching a root object each time (most likely the application), and navigating to the relevant low-level nested object to access it (Supplier, Contact, Position…). When a low level object requires updating, the entire graph needs to be sent back to the GemFire cache. This requires massive serialization and network overhead. When entities are shared between root objects, race conditions and data consistency problems result.

Having multiple entities within the same region can be catastrophic in terms of performance, because indexes are defined per region, so the more indexes there are, the more write performance drops. Data can be modeled using a nested graph, where each parent object contains its associated objects, but this means that every data retrieval on a nested object also requires retrieving the parent object, and any update to the child object would also require an update to the parent object. This can have huge performance implications.

Queries and transactions across regions can also cause major performance and consistency issues. Co-locating data from different regions into the same partition is also very complicated. All this makes it practically impossible to model a complex application, with multiple associated entities, using GemFire. You end up with an application that is totally locked into GemFire.

The Space – Natural In-Memory Object Store

GigaSpaces in-memory object storage technology is constructed from a global distributed shared memory store called the space. It can be compared to a distributed in-memory object database in which objects are categorized by class type, like a database table. Beyond enabling the writing and reading of data, the space provides rich event-based operations and ability to execute business logic with the data through map-reduce.

Applications interact with the space by writing and reading objects from the space through a space proxy. The client application does not need to construct a space per class, it simply uses a single proxy for the entire application where the space is deployed, usually in a partitioned backup topology (shared and replicas). Application objects from different class types (entities) can participate within the same transaction, be co-located within a specific partition, and persist using the same database transaction.

Going back to the job application example, with GigaSpaces you can have each of the entities (Supplier, Contact, Position, School…) stored as-is, within the same distributed space, each with its own indexes. Each can be updated without affecting any other and a transaction can span them all. Objects can reference each other through IDs, similar to a database key.

Because the space is object-oriented and aware, a query on a super class also spans its sub-class instances. Indexes defined on fields for class A do not affect write performance for objects
written from type B (A and B do not share the same super class). There is no need to model the data using the key/value approach, ensuring keys are unique, but simply write objects into the space as-is (POJOs). Advanced users can use a Document model (similar to document store databases like MongoDB) as the space internally supports unstructured (schema-less) data items. In addition, standard Java APIs such as JDBC, JMS, JPA, and even HashMap interface (JSR 107) are supported. All these are simply translated to the native space API. With such native and simple OO modeling, application and data model design are hassle-free, and there is no need to bind the design and architecture into a specific non-standard protocol.

Replication vs. Partitioning

Gemfire appears to be assembled from two separate products – one that is designed to support replication and another designed to support partitioning. It appears that each was developed by a different development team. There are many features in the replicated topology that are not supported in the partitioned topology, making the product more difficult to use in the partitioned model. It is not possible to move transparently from the replicated to the partitioned mode.

GigaSpaces does not impose such limitations. The partitioned mode is actually a variation of the replicated topology: For multiple replicas, there is actually only one partition, where there is one master and all other replicas are slaves or all the replicas are masters. The replication between the replicas can be synchronous or asynchronous.

Classpath and Data Versioning

Setting the cache node classpath with Gemfire is one simple example of issues that arise using GemFire in a production environment. When using GemFire for HTTP session management or any other type of application you must include the HTTP session classes or any other dependent classes your key/value object uses as part of the GemFire classpath. When using indexed data, the relevant classes must be part of the cache node classpath. In a small system this might not be an issue, but in an enterprise environment it would be: think about a 100 or 1000-node cluster. This would be a major issue. This also means that every application change requires shutting GemFire down to update every cache node classpath.

With GigaSpaces, this issue does not exist. Data object classes and their dependent classes are not required as part of the cache node classpath. These are dynamically loaded in a lazy-manner into the cache JVM classloader via distributed classloader technology that is fused into GigaSpaces remote invocation protocol. This means you can add new Data object types, and index them as you wish without any cache downtime.

When changing data schema (adding/removing fields from a data object) there is no need to restart the cache node as GigaSpaces supports schema-less objects.
Rebalancing – Does the System Maintain Consistency upon Failure?

GemFire offers the classic buckets approach to support rebalancing. The question is whether it really works and supports real-life scenarios, such as maintaining data consistency when a distributed transaction is executed and a failure occurs, or when the rebalancing process triggers and there is insufficient available memory to accommodate the data. In our observation, it simply does not work. Data within the cache reaching inconsistent state with these scenarios.

GigaSpaces rebalancing technology designed to deal with extreme cases where transactions are executed while rebalancing happens or when there are failure during the rebalancing activity. The cache would never reach into inconsistent state.

Partitioned Regions with Redundancy

Why transactional operations will not work:

From GemFire *Partitioned Regions with Redundancy*:

> Write operations always go to the designated primary and get distributed to the redundant node. However, read operations are directed to primary as well as redundant copies, allowing a read intensive system to scale much better and handle higher load.

This approach, where backup instances are used also for read operations, has potential issues: What happens when a transaction is executed? Are both primary and backup copies involved in the transaction? If so, every transactional operation is conducted against both the primary and the backup, causing massive overhead. Any failure of the backup node means that the new generated copy need to also maintain the transient state of the transaction to enable it to roll back in case of transaction abort.

With GigaSpaces, read operations are always conducted from the primary copy. There is no access to the backup instance by the application. A transaction is executed against the primary and once committed is also committed against the backup. This ensures optimized network utilization, and full data consistency.

Capacity Planning

Can you plan the number of failures your system should be able to survive?

GemFire requires having enough capacity available at deployment to survive multiple cache node failures. For example, for a 100 GB cache spread across 10 JVMs, to survive three failures without human intervention it is necessary to have 13 JVMs up and running at the deployment time. In the event of cache node failure, the replacement JVMs must be started manually. However, what about a 1 TB system? Assuming each JVM has a 10 GB heap, this means 100
JVMs will be required to accommodate such capacity. How would you identify which JVM failed, and where you have available machines with spare memory to start a new cache node?

From the GemFire documentation:

**Planning for Enough Members to Support Redundancy**

Most of the setup for redundancy is in figuring out what you need to provide adequate capacity for your region data. If you need more space at run time, you can start a new member host.

Make sure you start enough members to provide the level of reliability you require. To calculate the number of members you need for redundant data storage, assuming that all your members are on separate machines, use this formula as a general guideline:

\[
data_{store\_members} = redundant\_copies + 1 + concurrent\_member\_failures
\]

where:

- \(data_{store\_members}\) = members with caching enabled that participate in the partitioned region
- \(redundant\_copies\) = value of the redundant-copies partition attribute
- \(concurrent\_member\_failures\) = number of inaccessible members that your partitioned region must be able to tolerate at any given time without data loss

With GigaSpaces, the capacity planning exercised is very simple. You define in the GigaSpaces system the SLA you want to maintain, and deploy. As long as there are available resources, GigaSpaces will automatically maintain this SLA.

Here is an example for 1 Tera byte cache cluster deployed SLA where the initial size is 128GB:

ProcessingUnit pu = gsm.deploy(
    new ElasticSpaceDeployment("mySpace")
        .memoryCapacityPerContainer(16,MemoryUnit.GIGABYTES)
        .maxMemoryCapacity(1024,MemoryUnit.GIGABYTES)
        //initial capacity
        .scale(
            new ManualCapacityScaleConfigurer()
                .memoryCapacity(128,MemoryUnit.GIGABYTES)
                .create())
);  

With the above command, GigaSpaces will start a cache cluster across the available machines, calculating the right number of partitions at the time of deployment. If the system receives a scale trigger (based on memory, CPU, or custom user logic), new cache container (JVM) will be started and partitioned will be rebalanced. In the event of failure, a new cache container is started, accommodating the missing partition. For more information see Elastic Processing Unit.
Cache Configuration

GemFire offers no way to have a centralized cache configuration. Every cache node must have a local copy of the cache configuration. There is no central deployment functionality enabling the user to hand the cache cluster configuration into a deployment server entity and push it to all cache server nodes. For a cluster of with a large number of nodes – say 100 or 1000 nodes – just dealing with the cache node configuration would be virtually impossible. With a dynamic environment such as a cloud or virtualized environment, this raises significant questions as to the usefulness of this choice.

GigaSpaces enables you to deploy the cache server into an unlimited number of machines without copying any configuration files around, or accessing any of the machines running the cluster. Using a simple command or API call, all cache nodes are provisioned and configured. An elastic manager provisions new VMs (when running on the cloud) and a deployment manager (GSM) starts the cache containers and pushes cache cluster configurations and relevant application binaries to all containers (GSC) hosting the cache cluster. Multiple cache clusters can be deployed into the same container grid with total separation between them. GigaSpaces allocates a new classloader for each deployed cache instance within the container (JVM) enabling you to have real multi-tenancy, store multiple versions of the same class within the same JVM, and undeploy cache clusters gracefully.

System Recovery from Failure

Does Cache Process Failure Mean Entire System Failure?

With GemFire, if a cache node fails, there is no recovery mechanism. If there is enough spare memory capacity across existing cache nodes, data integrity will be maintained. However, if the system lacks available memory within the cache nodes, specified redundancy behavior does not occur, causing potential data loss the next time a failure occurs.

Returning to that example cluster of 100 or 1000 nodes: If one or more nodes fail, locating the correct machine with the necessary memory or CPU resources to start a new GemFire cache node can be a very complicated task. There is no notion of SLA, or of maintaining a predefined SLA during the life cycle of the cache cluster. When a cache node fails, it is necessary to find the relevant machine with the right resources and start a new cache node (JVM) manually.

With GigaSpaces, in the event of container (JVM) or machine failure, GigaSpaces automatically locates a machine with free CPU and memory resources, starts the container, and provisions the missing partition (and its co-located business logic if there is such) into the newly-started container. This approach enables the system to maintain the cache cluster SLA capacity specified at deployment time – without any human intervention. When the cache needs to scale, GigaSpaces starts new containers dynamically and rebalances the cluster. In a cloud
environment, a new VM is started on-the-fly, enabling the system to consume new resources on-demand and later free these resources when the system needs to scale down.

Disclaimer

This benchmark was designed to provide initial performance numbers for basic cache operations. It is intended to provide the reader with comparison information between GigaSpaces XAP Caching Edition and vFabric GemFire. Neovise utilized outside technical resources to run the tests and produce the results. Interested readers are invited to download the entire benchmark code from github to see firsthand the differences in performance between the tested solutions.

This document is not intended to provide a comprehensive comparison between GigaSpaces XAP and GemFire or to provide information about the maximum performance each product might deliver. GigaSpaces XAP is primarily an application platform for real-time applications with a high-speed transactional in-memory data grid, whereas GemFire is a classic cache product.
About Neovise

Neovise is an IT industry analyst firm focused on cloud computing. Based on independent research and analysis, we deliver essential knowledge and guidance to cloud-related technology vendors, service providers, systems integrators and investors, as well as business organizations seeking to purchase cloud-related services and technology. Our services include research, advisory and collateral development that helps our customers—and their customers—make optimal decisions and formulate winning strategies. Research. Analyze. Neovise.

For more information, please visit www.neovise.com.

©2012 Neovise, LLC. All Rights Reserved.