

Considerations in Sizing a GigaSpaces Deployment

Peter Coates

December 7, 2007

This document contains some informal notes on general considerations for sizing a GigaSpaces deployment. These notes are intended to help identify the major factors that affect the amount of hardware you need, and to supply some tools and rules of thumb for estimating these requirements.

Many of the issues discussed here apply to any pure-Java program, not just to GigaSpaces.

1 Introduction

This document attempts to pull together a summary of the many factors that determine the hardware/software configuration required to support a GigaSpaces application. It's a complicated subject not only because so many influences contribute to the memory and CPU demands of a program, but because the potential trade-offs are so numerous: latency vs. throughput, reliability vs. latency, performance guarantees vs. total cost... The list is long.

As there are too many permutations for there to ever be a single useful formula, instead, we merely enumerate some the many factors that can affect the final decision, and try to provide some guidance on how to estimate some useful numbers. Among these factors are:

- Deployment topology.
- The size and instance count of objects as well as their life-cycle patterns, indexing, etc.
- Tolerance limits of your program for latency, variance, reliability, etc.
- The VM implementation and tuning.
- OS choice and configuration.
- I/O and computational requirements.

2 What is a Deployment?

A GigaSpaces application can be deployed across any number of partitions¹ in any of several different topologies.²

There are two aspects to deployment: the logical and the physical. A logical deployment is independent of the number of machines on which it is running, but generally remains constant between shutdowns.

The physical deployment (including the number of CPU's) can change dynamically, either because of failure, because machines are brought down deliberately, because more machines are added to share the load, or because the GSM elects to move partitions or services in accordance with SLA guarantees.

Some of the most important parameters of a logical deployment with respect to space consumption and sizing are:

- The cluster topology. Variants include: singleton, partitioned, fully-replicated, and the use of local caching (these are also spaces). The number of partitions is chosen to meet the expected demand for any combination of:
 - Memory
 - CPU
 - I/O, threading, or other capability supported by hardware.
- Backup policy. Backing up a space multiplies the memory consumed by the application by one plus the depth of backup. Synchronous backup adds relatively little to CPU usage in most cases, but can increase latency significantly compared to non-backed up usage.³
- Persistence policy, e.g. mirror space, synchronous writes, RDBMS vs. files, etc. In some cases the persistence policy is “none.”
- Granularity of the objects stored, indexing, and other properties of the application.

¹Both individual spaces and federations of spaces are often (confusingly) referred to as spaces. A space spanning multiple VM's is a federated collection of individual spaces which individually behave much as they would without federation.

In such a configuration, the individual spaces are called “partitions.” The set of data residing in such a federated space is partitioned in the mathematical sense, because the individual spaces hold subsets that are mutually exclusive and collectively exhaustive.

²The most common topology is called “partitioned sync-to-backup,” in which data is divided among several sub-spaces, known as partitions. Each of the partitions has one or more sister spaces that redundantly hold exactly the same data to protect against loss in the event of a failure or deliberate shutdown of a partition. Each of the sisters (usually there is only one per partition, but it can be any number) is synchronously updated with any space-changing operation that affects the corresponding primary. Ordinarily, the services on a backup partition do not run until/unless it becomes a primary.

³The increase in latency is still small compared to that incurred by backing up to a relational database. A variant known as sync-rec-ack yields nearly as much safety as full synchronous replication by allowing processing to proceed once the modification has been committed to the network, rather than after the remote backup actually replies.

- Processing logic running in the space can be significant. Large numbers of temporary objects created in the space’s VM have space and performance implications (as described elsewhere in this document.)

Space containers are basically VM’s running a small amount of code to support federation, ServiceGrid functionality, SLA behavior, etc. Spaces are Java services that usually run in a container. They often (but need not) run one-to-a-container⁴ A given container may support zero, one, or multiple spaces, and/or, other services.

Just as spaces and containers need not be one-to-one, deployment of the containers on hardware CPU’s (or cores) also does not need to be one-to-one, and often is not. For instance, a large server often hosts far more memory per CPU than a single VM can support.⁵

Thus, more containers than CPU’s may be allocated to take advantage of available memory. More containers than spaces may also be allocated.⁶ When choosing hardware, it is not necessary to support the maximum future load—a suitable logical deployment can be chosen and scaled up or down physically as needed without changing the logical configuration or indeed, without even stopping the application.

3 Pure Java Object Sizes

The largest factor in the deployment footprint of a GigaSpaces application is usually (but by no means always) the size of the Java objects in the space. Whether using GigaSpaces or not, developers are often surprised by how much memory a Java⁷ program uses. Java expends a lot of memory to eliminate the complexities of explicit memory management and to simplify other aspects of

⁴A space can also run embedded in an ordinary Java program. Without the container spaces give up some services for which container support is necessary.

⁵Virtual memory is one of the reasons that configuration is as much art as science. Virtual memory multiplies the amount of memory that is effectively available by storing seldom used blocks of memory to disk. This slight of hand gives any number of processes the illusion of having the largest possible memory space, which is good, because there is much more memory available, but bad because it makes the amount of memory you can use *effectively* much more application-dependent. This is not a GigaSpaces or even a Java thing: it is at the OS and hardware level.

For virtual memory to be efficient, a program must have “locality of reference,” which means that needing a given stored value implies a high probability that you have recently used another value that was stored nearby in the physical memory (making it probably that your data has already been paged in with it.) Without the presumption of locality of reference, for applications of reasonable size, almost every memory access would involve a physical disk read.

For obvious reasons, an application with many distributed workers accessing a large shared space is likely to have relatively less locality of reference than would occur in an equivalent non-distributed program. This leads to subtle interactions between total memory required, memory usage, number of workers, application behavior, and even data-structures.

⁶This is often done to give spaces a place to migrate to under SLA, or to have idle containers ready in case new backups need to be started, etc. Another reason is to allow the GSM to deploy processing units that host services which are not co-located with a space.

⁷The same applies to most similar VM-based language such as C#.

programming. As will be seen below, in a typical application, language overhead is usually greater than the size of the raw data.

Before getting into GigaSpaces-specific memory requirements, let's look at what objects cost in any pure-Java program.

The nominal memory requirements for Java primitives are:

Primitive	Bits
boolean	1
char	16
byte	8
short	16
int,float	32
long,double	64

These types, however, are only guaranteed by the compiler to *act* like they are of their nominal size; the actual sizes of the primitives and the Object versions of the primitives are always much bigger for a variety of reasons. The details vary from platform to platform but the following issues contribute to the overhead.

- While the byte is the lowest logically addressable unit, machines actually move data around in machine words⁸.
- Within objects, the primitive types are usually stored on word-aligned addresses, wasting the difference between data size and word size. This is done to simplify compiler operations.
- Each object has hidden overhead for tracking, usually 8 bytes.
- The compiler decides how immutable objects (e.g., Strings, Integers) are allocated and stored. Identical immutable objects may be stored uniquely and referred to from many places, or they may be stored many times. They are only guaranteed to be logically immutable.
- There is an unspecified memory overhead that is not part of an object instance (for garbage collection and other purposes)
- The VM itself needs room to move—you can see how much by overloading a VM from an ordinary Java program and creating and abandoning a lot of objects. Long before an out of memory error occurs, you will see long and frequent GC delays.⁹

Java (Unlike C and C++) does not provide a good way to compute the size of an object from within the language. Unfortunately, even a function like the

⁸Hardware for a given machine—the bus, registers, TLB's etc.—are built to the word size. Instructions usually exist for manipulating bytes, but compilers tend to choose instructions that move data the fastest regardless of space-efficiency.

⁹The recommended maximum heap usage is 60%-70%

sizeof operator would be of only limited help, because with garbage-collected languages object size and number of object instances alone are not sufficient to compute the total memory requirements of a program execution. The rate of object creation and destruction, the granularity of objects, and a host of trade-offs enter into the picture.

The notion of being “out of memory” also tends to be less well defined than it would be with a C++ program. While it is possible for virtual memory interactions of a C++ program to result in page-thrashing, being out of memory tends to be much more cut-and-dried than it is with any garbage-collected language. For instance, C++ has no concept of GC pausing, nor is there any analog to the memory thrashing that occurs as a garbage collected language runs short of space. And of course, like any other program, a Java VM memory usage is subject to all the ordinary frailties of virtual memory.

3.1 Three ways to Determine Object Size

The following three work-arounds for computing object size are often cited in the literature.

3.1.1 Serialization

Taking the size of a serialized representation is often recommended for estimating memory footprint, but does not appear to work very well. Don't bother.

3.1.2 Empirical Measurement

This method relies upon measuring the memory consumed by an application before and after creating a large number of objects. The basic steps are:

- Invoke garbage collection several times.
- Query the runtime for memory consumption.
- Generate a large number of instances of the object you wish to measure the size of. Bear in mind that the cost of Strings may be highly variable depending upon whether or not they are re-used. Some VM's detect re-use of a String and refer to the same copy repeatedly. Others simply create a new one.
- Invoke garbage collection several more times.
- Again query the runtime for memory consumption.

Sample code for doing this is included in Appendix A.

3.1.3 Calculation

Calculate object size based on the following table of sizes of simple objects. Most String sizes are omitted for brevity.

Object Type	Nominal Size	Actual Bytes
Integer	4	16
Float	4	16
Double	8	16
Long	8	16
Short	2	16
Byte	1	16
String	0	40
String	1	40
String	10	56
String	30	96
String	40	120
reference (32)	4	4
reference (64)	8	8
array types	0	16

Note that the cost of an array is 16 bytes above the space allocated for primitive types or references. An array of arrays incurs this expense for each of the inner arrays, as well as for the outer array. Thus, *byte [[[arrayOfArrays = new byte[50][2]* which one might imagine would take 100 bytes, might actually take 1216 bytes or more on a 32-bit machine.¹⁰

4 General Considerations Affecting Size

GigaSpaces is a very general platform for deploying a distributed application. It supports a host of application architectures, so of course there are a bewildering array of options for topology, replication strategy, access patterns and data objects. Estimating hardware requirements can be confusing. The following is an enumeration of some of the considerations that can affect the hardware requirements.

4.1 Memory Requirements

Object size is affected by a number of considerations, the biggest of which is the heavy use of memory by Java itself.

See section *Java Object Sizing* for details on sizing Java objects.

¹⁰The two byte arrays would be word aligned, and each two byte array has an overhead of 16 bytes. Each of the references is four more, and the outer array itself has an overhead.

- **Native Java object size:** Java objects are often much larger than one would intuit.
- **Indexing:** GigaSpaces provides two basic options for indexing.
 - Default indexing is hash based. Each index costs about 250 bytes per entry in the space.
 - More powerful B-Tree indexing can be declared in configuration. B-Trees reduce the time required to do lookups and range queries at the expense of increased time to insert and substantial space expended per-entry. A B-Tree index costs about 500 bytes per entry.
- **Compression:**
 - Zip-style compression is available as a configuration option. This gives a modest amount of compression at the expense of more CPU consumption at each end.
 - Only fields necessary for space-lookup operations need to be exposed at the top level of an object. Data which is used by services or end users need not be exposed, and thus need not incur the expense of meta-data in the space.¹¹ Boxing all such fields into a nested object can result in substantially less memory being used. There is no runtime penalty for saving space by coding in this way.
 - Objects with numerous small fields can often be greatly compressed by “byte array compression.” High degrees of compression are often obtained.¹² This technique (which is supported by the product) uses a byte array to provide storage for Java objects, avoiding the substantial space wasted by byte-alignment and other lower level costs, as well as allowing one to take advantage of known properties of the data domain.

Fields are stuffed into an array of raw bytes. The access positions for a given datum are stored statically, i.e., only one copy per *class*. This method requires explicit *pack()* and *unpack()* methods be executed (at store and read time, respectively.)

There is some runtime penalty both in CPU and in the amount of object creation (and the resulting garbage collection) for use of this style of compression, but the operations are not onerous, Much less data must be stored and moved, meaning that in practice, one can expect overall performance to improve, not decrease.
 - General Java techniques for economizing on object size are useful. For instance, Java Strings use two byte characters by default—UTF-8 encoding reduces this cost by half.

¹¹What is the cost of an exposed field?

¹²The biggest savings tends to come in objects with lots of small fields. In the financial industry, such objects as trades and instruments often have scores of fields and get very good results.

4.2 Application Behavior

A non-exhaustive list of application behavior that affects space includes the following.

- **Granularity:** The space exacts a constant per-object overhead. Storing many small objects costs much more than storing fewer coarse-grained objects.
- **Latency:** Java garbage collection delays increase out of proportion to VM size. Thus, for low-latency applications, smaller VM's will have smaller average latency and a lower variance.
- **Read vs. Write behavior:** Applications which use read-mostly data can tolerate larger VM's because of the lower garbage generation.
- **Backup:** One level of backup doubles the memory required. Two levels triples it, and so on.
- **Persistence strategy:** Asynchronous persistence gets the database and/or disk writes out of the critical path, enabling low-latency and high throughput. However, it is important to remember that the critical operations must still be held in the space until written to the RMDBS, and application architecture must allow for this. If sustained throughput exceeds the rate at which disk writes can occur, then space requirements can increase without bound.¹³

4.3 OS and Hardware Platforms

Platform and operating system affect GigaSpaces deployment.

4.3.1 Thirty-two Bit Systems

Platforms with a 32-bit word size have a nominal four GB virtual address space (2^{32} or 4,294,967,296 bytes) of which only some is available to user processes.¹⁴

¹³There are strategies for getting around this, for instance, logging to file instead of database. Few databases can sustain thousands of writes per second, but file writes can achieve almost any required density.

¹⁴The address space limit applies *per-process*, not per CPU. This is because processes do not actually use the machine addresses; they exclusively manipulate virtual memory addresses in a virtual memory space, of which there is one per process. Every address used by a program is mapped to a physical machine address through an elaborate set of data structures known as a page-table.

In theory, almost any number of processes can each have a virtual memory space of up to four GB, even on a platform with less than four GB of physical memory in total. In practice, the number of address spaces and how they behave under use is application and operating system dependent. Linux, for instance, allows one to choose whether page table allocation will fail-fast if an operation grabs too many pages to fit in the remaining page file space, or will optimistically ignore the purely notional use of pages that have not actually been touched yet, forbearing to fail until the limit is actually reached. There are many other tunable behaviors at the OS level.

See any operating systems textbook for details on how this magic works.

All 32-bit platforms have address spaces that are small relative to the potential size of a GigaSpaces deployment.¹⁵

- **Microsoft Platforms:** 2.0 GB are available to user processes.
- **Linux, Sun and other Unix-like OS's:** Amount available for user processes is configurable. It is almost always at least 2.0 GB and is typically more like 3.5 GB.

As a VM approaches the maximum memory available in a 32-bit VM they begin to spend undue amounts of time garbage collecting.

4.3.2 Sixty-Four Bit Systems

Sixty-four bit platforms have huge address spaces in theory, but in practice are limited to about 10.0 GB depending upon usage patterns. The amount of space that can be addressed effectively is a matter of both OS and VM implementations. This is improving as 64-bit environments mature.

Pointers on these systems consume eight-bytes each, not four, as in a 32-bit system. Thus, in the choice between 32-bit and 64-bit, one is trading off object size for total VM memory capacity. As multiple VM's can be used on a 32-bit machine to get a large space, a 64-bit environment is not as advantageous as might first appear.

5 Determining Deployment Size

Memory consumption in the space is affected by many factors. The cost for a given object is determined by the following:

1. Size of the underlying Java object.
2. A per-object constant overhead.
3. A size-dependent cost of approximately 50% of the object size.
4. Indexes: each ordinary index has a per-instance cost of about 250 bytes.
5. B-Tree indexes: each has a per-instance cost of about 500 bytes.
6. Longevity of objects in the space. The number of objects in the space is proportional to the creation rate multiplied by the lifetime of the objects.

Estimating memory requirements is pretty obvious:

- Use one of the Java size estimation techniques given above. It is worth while to verify ones calculations on at least one object type with an empirical test, which can be easily done from within JUnit or Eclipse.

¹⁵Not all of the “available” space is available for your code, of course. The VM, containers, etc., are also using this space.

- Multiply by 1.5
- Add the per-index cost.
- Multiply by the number of objects you plan to use in the space at once.

If your program is data-bound you can compute the number of machines directly by dividing the total memory requirement as computed above by the amount of memory you plan to have on each machine.

Choosing VM size:

- For a data-bound application, you can often run several VM's per CPU to get access to the memory.
 - On a Linux or Sun box, try VM's of 2.0 GB (max is about 3.6, configurable.)
 - On a Microsoft, try 1.0 to 1.5 GB per VM.
- A compute-bound application is harder to estimate for. As above, you can get a lower bound on CPU requirement based on memory, but you may want many more. Remember, you can always add more CPU's later. Most well designed applications scale linearly—try your deployment scaled for a single machine, even a smaller development box, to get a feel for the general ratio of CPU to memory.

Another important point about CPU-bound calculations is that you need not use a single model for big calculations based on the Master/Worker pattern. Your embedded services can be supplemented by similar services running in PUs that come and go. Properly designed stateless services that do only transactional operations against the space can be brought on line and shut down at any time.

- For a low-latency/high-throughput application, size the VM's to minimize garbage collection. Use 1.0 GB or less per VM.

5.1 Optimizing for Memory

Of course, the default memory optimization strategy should be “do nothing special.” If this doesn't do the trick, there is a lot you can do to shrink memory usage, and it is often easy to rake in big savings for a small amount of work.

One neglected reason to optimize memory consumption is to increase speed. Less data is faster for lots of reasons, but one big reason is that there's simply less data to move across the network links. For large objects, the speed increase can be inverse to the shrinkage factor, i.e., halve the object size and double the speed.¹⁶ Speed increases, in turn, often reduce the total data burden by lowering the time data is in the system.

¹⁶Your mileage may vary.

- Look again at what you are storing. The prospect of unbounded memory can make one forget to use it carefully:
 - Are you storing a huge business object when you only need a small subset of the fields maintained live?
 - Can you normalize out part of the data and refer to it rather than repeating it in-line?
 - Are you storing a week’s worth of data when all you need from the preceding four days is an aggregated summary?
 - Can you discard intermediate results that can be re-computed cheaply?
 - Is there a less costly representation?

Most applications have a lot of slack of this kind.

- The compression options given in Section 4.1 can result in large savings.
 - Byte-array compression usually works best for data with lots of small fields. A given field must be of consistent size, so this doesn’t always work for string fields which can be of varying length.
 - Concealing most fields in a nested object is usually the second most effective.
 - Default Zip compression works best for Strings.
 - The techniques can be used in combination: for instance, using byte array compression does not preclude also using Zip.
- Local caching can be a valuable option. In some applications and topologies, reference data can be stored centrally and only a small subset accessed from any given service via local cache or view.
- Pay attention to how long data survives in the space and when it is brought in. It makes sense to use read-through/write-through strategies for seldom-used data.
- Don’t automatically store intermediate results. There can be a useful trade-off between storing intermediate results and re-computing.

6 Useful Calculations

These subsections give some tips on calculating the number of objects in a space application and other useful numbers.

6.1 Little's Law: Number of Objects and Latency

Little's law relates the rate objects are inserted into a system, the amount of time they will be resident in the system, and how many objects will be in the space at one one time:

- Given the arrival rate of messages and the number of objects in a space, you can compute the processing latency.
- Given the latency and the arrival rate, you can compute the amount of space required.
- Given the latency and the amount of space available, you can predict the maximum throughput the system will support.

The law states that the average number of objects in a stable system, N , is equal to the product of their arrival rate λ and the average amount of time, T , that they are resident in the system.

$$N = \lambda T$$

For instance, if a computer system receives messages at the rate of $1000/sec$, and the average time required to dispatch a message is $20ms$, then there will be an average of 20 messages awaiting dispatch at any one time.

It can be even more useful the other way around, because small processing latencies are difficult to measure accurately, while object-count and arrival rate usually are not. For instance, if you discover empirically that the average number of messages in the system is twenty, and that the arrival rate is $1000/sec$ once may fairly calculate the processing latency as $20ms$.

Conveniently, the underlying object arrival distribution is irrelevant to the validity of Little's law. Despite the law being independent of distribution, identifying both the periods of sustained heavy load and the overall average load can be critical. This is merely to say that while a bursty distribution is not important, but that if a heavily loaded interval continues for long enough, it can become the relevant context when sizing for memory.

In such a situation it is not the *overall* average rate of insertions across a run of arbitrary length that matters, but the average over some particular *sub-interval*. Unfortunately, you can't assume that the maximum memory consumption will necessarily occur during the heaviest insertion load either—the pattern for the entire day should be examined.

Differentiating between spikes sustained peaks is critical also because, while spikes don't matter w.r.t. Little's law, they may be critical in other contexts (such as guarantees of maximum latency.) In many applications it is not *average* latency that matters, but *maximum* latency, latency variance, or some bound upon the percentage of messages that may exceed a given latency.

6.1.1 A Note on Latency

Latency requirements definitely affect deployment size and configuration in a number of ways, but Little’s law covers only *average* latency while often variance is equally or more important. An extremely low average may not be much help if it includes wildly expensive outliers due to garbage collection delays. It may be important to nail this down in the requirements—the distinction can go unnoticed until system test or even production! If variance as well as average is a concern, consider a real-time operating system and virtual machine implementation.

“Real-time” in the context of VM’s and Unix-like OS’s is a relative thing. What RT means in practice is a guarantee of at most λ microseconds for $n\%$ of the responses, and larger limit on latency for the other $(100 - n)\%$ of the responses. The larger limit is usually pretty tight, say, a couple or three milliseconds. Use of real-time Java with GigaSpaces is beyond the scope of these notes but the product has certainly been used with RT and information on this is available elsewhere.

6.2 Amdahl’s Law: Bang For the Buck

Amdahl’s law gives the limit to performance improvement that is possible when an improvement is applied to only part of a computer program. Intuitively, Amdahl’s law tells us that the improvement one will gain from a performance enhancement is limited to the domain over which the enhancement applies.

Two variations are given: the general form and a version that addresses parallelism in particular. The general form takes into account the possibility that a speedup covers several separate areas of the code and may not accelerate them all equally. The formula is:

$$\frac{1}{\sum_1^n \left(\frac{P_k}{S_k}\right)}$$

where

- P_k is the percentage of instructions affected by the k th speedup.
- S_k is the speedup multiplier of the k th speedup.
- N is the total number of speedups.

For example, if only half the instructions executed by a program are subject to improvement, then at best, one can only double the performance of the program no matter how amazingly effective is your speedup.

Note that this formula applies to the total instructions executed, not to the actual lines of code. For instance, if a program has 90 lines of setup, then a tight 10 line loop that executes a million times, the proportion that can be optimized is not 0.1 but 0.99999—an almost ideal case.

6.3 Amdahl's Law Applied to Parallelization

Parallelization is a special case of performance enhancement. Performance speedup due to parallelization of a program is given by the following formula

$$\frac{1}{F + (1 - F)/N}$$

given that

- F is the sequential fraction of the program.
- N is the number of processors to be applied to the problem.

Note, that in parallelizing a program, one is attempting to make F as small as possible.

6.3.1 Pareto Principle, A.K.A 80-20 Rule

Eighty percent of the X comes from twenty percent of the Y .

The application of the Pareto principle to execution time of computer programs is often attributed to Donald Knuth: eighty percent of the time is spent in twenty percent of the program. Note the relevance of Amdahl's law—don't waste time speeding up the eighty percent of the code that is lightly used.

6.4 Back of the Envelope

The following trick is worth remembering in all of these calculations: always write down every unit in a formula: milliseconds, objects, processors, whatever. The unit symbols may be cancelled and/or combined exactly as if they were variables, and the units remaining at the end should be the expected units of the answer: e.g., objects/partition, or ms/task, etc. This is a sanity check for your calculations—if the problem is formulated incorrectly, usually the units of the answer will be obvious nonsense.

7 Appendix A: Code for Measuring Object Size

The example below implements a simple way to measure object sizes in Java. To use it, incorporate the code below in a test routine as described and run against a space running in your own process. Alternatively, the code can be invoked from a service running on a remote space. The critical thing is that the code run in the same VM with the space.

- Run `startMemTest()` to force garbage collection and measure the memory before you start.
- Then run some code to insert a large number of objects of uniform type—you want to have a few hundred megabytes of objects in the space.

```

private static final Runtime s_runtime = Runtime.getRuntime();

private static long prevMemory = 0;

private static long postMemory = 0;

private static void endMemTest() {
    int size = lst.size();
    System.out.println("Saved total of:" + size + " records." + "");
    long memConsumed = postMemory - prevMemory;
    System.out.println("mem before:" + prevMemory + " after mem:"
        + postMemory);
    System.out.println("mem per record:"
        + (((long) memConsumed) / totalPublished));

    System.out.println("Records still pinned in memory:" + lst.size()
        + ". Is record null? " + (lst.get(lst.size() - 1) == null));
}

private static void startMemTest() {
    try {
        runGC();
        prevMemory = usedMemory();
    } catch (Exception x) {
        System.out.println("runGC() failed:" + x.getMessage());
        System.exit(1);
    }
    System.out.println("Space memory prior to sending records:"
        + prevMemory);
}

private static void runGC() throws Exception {
    for (int r = 0; r < 4; ++r)
        _runGC();
}

private static void _runGC() throws Exception {
    long usedMem1 = usedMemory(), usedMem2 = Long.MAX_VALUE;
    for (int i = 0; (usedMem1 < usedMem2) && (i < 1000); ++i) {
        System.out.println("Running finalization and gc...");
        s_runtime.runFinalization();
        s_runtime.gc();
        Thread.yield();
        usedMem2 = usedMem1;
        usedMem1 = usedMemory();
    }
}

private static long usedMemory() throws Exception {
    return s_runtime.totalMemory() - s_runtime.freeMemory();
}

```

- Then run *endMemTest()* to clean up any garbage.
- The method (*usedMemory()*) will return the total memory your objects are using.
- Divide this quantity by the number of objects you have inserted to get the object size.

Note, this general strategy can be adjusted to use the size of the old-gen pool, rather than the memory as a whole. For some purposes, this will give a better value for object size for estimation purposes.