

Coherence Planning: From Proof of Concept to Production

An Oracle White Paper
November 2008

Coherence Planning: From Proof of Concept to Production

| | |
|--|----|
| Introduction | 3 |
| Preparing to Test..... | 3 |
| Architecture | 3 |
| “Stretch Clusters” and the “Split Brain” scenario | 4 |
| Multi-Threading and Thread Safe Operations | 4 |
| Development Techniques | 6 |
| Performance and Scalability..... | 6 |
| Support Tools..... | 9 |
| Management and Monitoring..... | 9 |
| Sizing..... | 10 |
| How do you determine how big your cache needs to be? | 10 |
| How do you limit the size of your cache? | 13 |
| How many JVM's? | 14 |
| Configuration Management..... | 15 |
| Security | 16 |
| Change Control..... | 16 |
| How do I change my configuration on the fly?..... | 16 |
| How do I change my cache objects in a production system? | 16 |
| How do I keep my .NET, C++ and Java code synchronized? | 17 |
| Testing..... | 18 |
| Testing Strategies | 18 |
| What to test? | 18 |
| What to measure?..... | 18 |
| How to test?..... | 18 |
| Example Proof of Concept (PoC) Tests | 19 |
| Measuring Latency and Throughput | 19 |
| Scalability | 19 |
| Data Reliability..... | 19 |
| Destructive Testing..... | 19 |
| Deploying to Production..... | 20 |
| Transitioning from Test to Production | 20 |
| Roadmap..... | 20 |
| Conclusion..... | 22 |

Coherence Planning: From Proof of Concept to Production

Taking a Coherence *Proof of Concept* and making the environment production ready requires careful planning to ensure the full benefits of Coherence are realized.

The volumes and type of data will typically determine the caching scheme and the network topology will determine the layout of the cluster(s).

INTRODUCTION

Whilst setting up a development environment for Coherence is relatively trivial, planning and moving this into production requires considerable testing and careful consideration to ensure the full benefits of Coherence are realized. The following observations and guidelines are meant to supplement those outlined in the Coherence Production Checklist, Performance Tuning and Best Practices guides, not to replace them. These documents should be read prior to reading this document, as the contents of these documents are not going to be replicated here.

PREPARING TO TEST

Architecture

For virtually all data caching scenarios the distributed cache scheme (i.e. Partitioned Cache) is the best option, because it provides much better scalability and is suitable for a wide range of use cases – read-only, read-mostly, read-write, write-mostly and write-intensive. Replicated caching effectively relies on scaling up, not out, to provide greater capacity, which can make replicated caches much more expensive from a resource (i.e. memory perspective) on a JVM by JVM basis. Similar performance for reads can also be obtained in many cases by using a near cache in conjunction with a distributed cache.

The level of resilience required by applications is also worth considering when planning your architecture. Not all applications require caching to be resilient to JVM failures, i.e. the data is easily re-creatable from a separate data source if the cached data is not configured to be redundant and a cache node crashes. If backups are not required, more memory will be available and write performance will be improved as it will only take 2 network hops for an update or insert, instead of 4 (2 to the primary and 2 to the backup).

For example, if no backups are used an insert only results in 1 network hop from the client to the cache node where the primary copy of the data resides and 1 network hop back to the client. If backups are used then before the response can be sent to the client the backup copy of the primary data on another cache node

needs to be updated too, resulting in 1 hop to the backup node and 1 hop back, or 2 additional network hops.

“Stretch Clusters” and the “Split Brain” scenario

To maintain a stable and resilient cluster Coherence relies on fast and reliable network connectivity between cluster nodes. Like all clustering technologies this is crucial to maintaining a single logical cluster. If a cluster is configured to span servers separated by a MAN or WAN as a “stretch cluster”, where the connection is relatively slow, e.g. several milliseconds, or un-reliable, the cluster may operate less predictably and the cluster can even break in two, if the connection is dropped for a prolonged period of time. This is called a “split-brain” scenario, where each part of the cluster at the different sites believes the other site has failed and that it is now the only surviving part of the cluster.

When this happens each part of the cluster will continue to function separately as a separate cluster. If the link returns then the cluster that is determined to have priority will take precedence and the other cluster will throw away its data and then join the more senior cluster. Seniority of the clusters is based on the number of active members in each cluster at the time the two clusters re-establish communication

The recommended approach to handling unreliable and/or high-latency networks is to replicate changes between sites using the Coherence Extend mechanism and/or the Coherence patterns for messaging. If data is already being replicated, data may just be re-read from the replicated data store. Finally the problem can be avoided by not deploying a “stretch cluster” unless the network connection is both low latency and considered completely reliable.

Multi-Threading and Thread Safe Operations

As Coherence is a multi-threaded environment its important that any application logic is aware of its threading rules. Operations performed on the service threads on cache nodes must take care not to cause a deadlock scenario. Service threads in Coherence may invoke:

- `MapListener`'s
- Network Filters
- Custom Serialization-De-serialization (e.g. `ExternalizableLite`)
- `BackingMapListener`'s
- `CacheStore`'s
- Query logic such as `Aggregator`'s, `Filter`'s, `ValueExtractor`'s and `Comparator`'s
- `EntryProcessor`'s
- `MapTrigger`'s

- `InvocationService` `Invocable`'s

So these should never make re-entrant calls back into their own services.

Service threads can have the following characteristics:

- A service may have three types of threads, a listener thread, a primary service thread and an optional thread pool. All of these can be thought of as "service threads".
- Refresh-ahead and write-behind caches use a special thread to invoke `CacheStore`, but these threads should also be considered to be service threads (e.g. in the event of eviction in a write-behind cache, `CacheStore.remove()` will be called by the actual service thread rather than the write-behind thread).
- Not all services have all types of threads.
- In many cases, a service thread *may* be the thread to invoke a piece of application logic, or it may end up being an application thread.
- Coherence does not always fail-fast when making re-entrant calls.

As an aside, it is especially critical to never spend much time in a client-side `MapListener`, as there is never more than one of them for a given service.

A *service* is defined as a unique combination of service type (e.g. `Invocation`, `Replicated`, `Distributed`) and service name. So you can call from a service "Dist-Customers" into one named "Dist-Items", or from "Dist-Customers" into "Repl-Inventory".

Service names are configured in the cache configuration file using the [services element](#). Whether the *call* is local or remote is irrelevant (in the current implementation). In particular, this complicates the use of key association to support efficient assembly of parent-child relationships.

If you use key association to collocate a Parent object with all of its Child objects, you cannot send an `EntryProcessor` to the parent object and have that `EntryProcessor` "grab" the (local) Child objects, even though those Child objects are already in-process. You can use direct access to the server-side backing map (which requires advanced knowledge to do safely), or you can run the logic on another service (e.g. `Invocation` targeted via `PartitionedService.getKeyOwner()`), and have that service access the data via `NamedCache` interfaces, or you can place the Child objects on another service which would allow re-entrant calls (but incur network access since there is no affinity between partitions in different cache services, i.e. the partitions could be on separate JVM's).

Using the `Invocation` service approach is probably the best compromise for most use cases (or embedding the Child objects in the Parent cache entry). Even when re-entrancy is allowed, you need to be very careful to avoid saturating the thread

pool and causing catastrophic deadlock. For example, if service A calls service B, and service B calls service A, there is a possibility that a sufficient number of concurrent calls could fill one of the thread pools, which would cause a form of deadlock. As with traditional locking, using ordered access (e.g. service A can call service B, but not vice versa) can help.

To summarize:

- A->A is never allowed
- A->B->A is technically allowed but is deadlock prone and should not be done
 - A->B and B->C and C->A is similarly restricted
- A->B is allowed
 - A->B and B->C and A->C is similarly allowed

Further information on Coherence cluster services can be found [here](#).

Development Techniques

Ensuring that developers continually test against a representative environment is also crucial so that expectations are set properly about performance, i.e. taking into account network hops, so there is early visibility of any issues related to performance and concurrency, etc.

A variety of Coherence features and development techniques can significantly increase both the performance and scalability of an application

PERFORMANCE AND SCALABILITY

Coherence scales linearly to provide predictable performance as data volumes and client requests increase. Cache sizes can be 100's of GB and it can support 1000's of users. Where CPU intensive operations, like cluster-wide events (`MapListener`'s, `NearCache`'s, `ContinuousQueryCache`'s) or cluster-wide requests (those targeted with `Filter` expressions rather than with one or more primary keys) such as queries and aggregations, a higher CPU:MEM ratio may be required.

In newer releases of Coherence (3.4+) objects can be stored in [Portable Object Format](#) (POF) and POF-native services can be configured. This can provide a real performance boost, as it enables caches to take objects in POF format and store them in the cluster without any intermediate transformations and vice-versa. Hence, the de-serialization and serialization steps will no longer occur in the proxy service and its CPU requirements will be significantly reduced.

The thread-pool size ([thread count](#)) configuration for a service can also impact scalability and performance, and can sometimes be counter intuitive. For instance, if more clients want to access data in a cache it would seem logical to just increase the number of threads in the thread pool associated with the cache service, thereby increasing the number of threads available to handle requests. However, this is not always the case, because the 'hand-off' from the service thread to another worker

thread is actually quite expensive. For short operations that don't involve any IO it is often more efficient for the service thread to process all the requests. Increasing the thread pool size will help increase the throughput and performance of a cluster if cache operations involve any network or disk IO, which will inevitably involve threads entering a 'wait state', while waiting for a response. Where a cache store is used or Entry Processors or Aggregators are performing network IO, the number of threads should be increased. The optimum number of threads will vary, depending on the characteristics of the data and operations being performed and should be determined by performance testing.

Although Coherence runs successfully on a wide variety of hardware, OS's and JVM configurations, not all are equal. For instance, Garbage Collection (GC) can take its toll on performance and a JVM like [JRockit Real-Time](#), which has deterministic garbage collection, could make the behavior of Coherence, from a GC perspective, more predictable. Likewise, different hardware and OS combinations should be considered, as some can be significantly better than others.

If you have or think you have ['hot data' or 'hot spots'](#)— that is data that there is a lot of contention for - then you may improve performance and scalability by re-factoring your object structure. 'Hot data' can unevenly balance the network traffic, directing more traffic to some nodes than others. To determine if you have 'hot data' you can run the [HotKeysAgent](#) over your cache. It will provide feedback on access traffic by cache server and object key. See the [Best Practice](#) guide for more information about 'hot data' and strategies to deal with it.

To maximize performance, from a development perspective, a wide range of techniques can be employed to tune Coherence. Although this document is not intended to be a development guide some techniques to consider are:

- Choose the [optimum serialization technique](#), usually POF in Coherence 3.4+. One set of tests on a low end machine produced the following results (based upon serializing and de-serializing 100k objects with a variety of fields and a nested child object):

| Test | Serialization Time (ms) | De-Serialization Time (ms) | Size (bytes) |
|---------------------|-------------------------|----------------------------|--------------|
| Java | 2360 | 10078 | 867 |
| ExternalizationLite | 484 | 1625 | 309 |
| XMLBean | 734 | 2070 | 322 |
| POF | 547 | 1234 | 186 |

- Co-locating related objects using [partition affinity](#). If you have related objects that depend or interact with each other then use [KeyAssociation](#) to ensure that they are co-located in the same partition and so the same JVM. This will eliminate any inter-object network traffic from read operations.

- When using a query that runs with a `Filter`, but without a `KeyAssociationFilter` (or in 3.4, a `PartitionedFilter`), it will be evaluated on each and every cache server. For a trivial query, that returns (say) 5 rows, this means that the actual query processing is trivial and doesn't benefit from scale-out, but the communication overhead grows linearly with the cluster size. For a sufficiently trivial query workload, the throughput of the cluster will be the same at 100 nodes as at 10. Even worse, queries will block until all nodes return, meaning that performance will actually decrease as the cluster grows. On the other hand, queries that are sufficiently compute-intensive will scale linearly and show constant performance. Most queries fall somewhere in-between. It's all a question of the compute-communicate ratio.

Note: Using a `KeyAssociatedFilter` to wrap (enclose) other filters will instruct the distributed cache to apply the wrapped filter only to the entries stored at the cache service node that owns the specified host key.

- Check that all queried attributes are indexed – using a `profiling filter`.
- Always use `putAll()` and not `put()` to put objects into a cache (using `Collections.singletonMap()` if necessary) as `put()` returns the old value across the network.
- Consider using some of the Coherence design patterns or common code. This has been reviewed, tested and tuned, so may improve the performance of your own code.
- Using `'lite' events`, where the old value of the object is not returned will reduce the amount of data being sent across the network.
- Accessing the value of indexes, rather than the object itself in an entry processor, can save the overhead of de-serialization the whole object, e.g.

```
import com.tangosol.util.processor.AbstractProcessor;
import com.tangosol.util.InvocableMap;
import com.tangosol.util.extractor.ReflectionExtractor;

/**
 * @author jhall
 */
public class GetAgeProcessor extends AbstractProcessor {
    private boolean liteTouchGetAge;

    public GetAgeProcessor(boolean liteTouchGetAge) {
        this.liteTouchGetAge = liteTouchGetAge;
    }

    public Object process(InvocableMap.Entry entry) {
        Integer age = null;

        if (entry.isPresent()) {
            if (liteTouchGetAge) {
                age = (Integer) entry.extract(new
                ReflectionExtractor("getAge"));
            }
        }
    }
}
```

```

        } else {
            // Normal mechanism to get the age
            Person person = (Person)
entry.getValue();
            age = person.getAge();
        }
    }
    return age;
}

```

- Storing application data in the binary format it understands, so removing data transformations (if the application is going to be the only consumer of the data).
- Keeping cache objects in a dual serialized/de-serialized format, with a custom `BackngMap` and `Filters`, to enable very fast object aggregation – as the custom `Filters` access the un-serialized data.
- If clients are connected to the cluster (via a proxy) over a relatively slow connection, say 100 Mbit, then performance may be improved by using the new [compression filter](#), to reduce the size of data retrieved by the client. However, in most cases the CPU overhead of the `CompressionFilter` will outweigh the bandwidth savings, so do not use it indiscriminately.
- When complex filters are used, try and merge many [EqualsFilter](#)'s into an [AllFilter](#) and many [OrFilter](#)'s into an [AnyFilter](#), to reduce the number of filters.
- Finally isolate the optimized settings for a specific cache using a separate service configuration in the cache configuration file. This will allow each cache to be tuned separately.

For more information on performance tuning see the [Performance Tuning](#) guide.

Use the same monitoring and management tools in the test phase to ensure that the results can be replicated in a production environment.

Support Tools

Management and Monitoring

Although the choice and testing of operational support tools is sometimes restricted to production environments, it's very important to ensure that the management and monitoring tools that will be used in production are also tested as part of the test cycle. Ensuring that these overheads are included in the test metrics means it should be possible to replicate them in a production environment.

A number of management and monitoring tools can be used to start and stop Coherence. These include:

- [WebLogic Operations Control](#) – to start and stop Coherence cache nodes.
- Shell Scripts.

- Windows Services.
- [Tanuki](#) – Java Service tool.

A combination of the above can also be used together along with other tools not listed. For monitoring Coherence the following can be used:

- [WebLogic Operations Control](#) – for dynamically and automatically managing SLA's and QoS. This can include starting new cache nodes when the existing cache nodes reach a memory or CPU usage threshold.
- [JRockit Mission Control](#) – JVM monitoring tool for low-level JVM performance analysis, even in production environments (its overhead is only 2-3%).
- [Oracle Application Diagnostics For Java \(AD4J\)](#) – provides production JVM diagnostics information for a range of JVM's (again imposing only a 2-3% overhead). In addition it can also provide database transaction information, like the SQL being invoked.
- JConsole – for Coherence JMX information.
- [JMX Reporter](#) – feature of Coherence 3.4 that allows Coherence runtime diagnostics information to be output to reports that can be customized for external consumption.

This is not an exclusive list and other 3rd party tools can also be used to manage a Coherence cluster. A key factor in choosing which management and monitoring tools are best-suited to your environment will be the type of information you want to capture and how you want to be notified. Many management and monitoring tools, like Oracle Enterprise Manager, IBM Tivoli and BMC Patrol, can act as a JMX container to allow Coherence JMX information to be displayed and used, or can receive and report SNMP traps generated by tools like WebLogic Operations Control.

It is very important to try and size the data grid caches accurately to ensure there is sufficient capacity for the expected data volumes.

Sizing

When trying to determine how much hardware you need for your Coherence cache a good starting point is to determine how much data you need to cache. Once this has been calculated you can then determine how many JVM's, what physical memory, CPU's and servers you need. These guidelines will allow you to plan your initial hardware requirements for testing. An accurate view of your hardware requirements can only be validated through specific tests that take into account your application load and use cases that simulate expected users volumes, transactions profiles, processing operations, etc.

How do you determine how big your cache needs to be?

To size your cluster a number of factors need to be taken into account. As a general rule you need to allocate at least 3x the physical heap size as the data set size –

assuming that you are going to keep 1 backup copy of primary data. More accurately the size of a cache can be calculated as follows:

$$\text{Entry Size} = \text{Serialized form of the key} + \text{Serialized form of the Value} + 150 \text{ bytes}$$

Hence:

$$\text{Cache Capacity} = \text{Number of entries} * 2 * \text{Entry Size}$$

For instance if a cache contains 5M objects, where the value and key serialized are 100 bytes and 2k, respectively:

$$\text{Entry Size} = 100 \text{ bytes} + 2048 \text{ bytes} + 150 \text{ bytes} = 2298 \text{ bytes}$$

Hence:

$$\text{Cache Capacity} = 5\text{M} * 2 * 2298 \text{ bytes} = 21,915 \text{ MB}$$

An estimate of the JVM heap space your cache data required can be verified by using the heap monitor in JConsole as well as by adding JMX monitoring to Coherence, so cache statistics can also be viewed. These cache statistics can be accessed in the following way:

1. To measure the memory requirements for your production installation start a cache server and load a portion (for instance 10%) of the largest number of objects you expect the cache to hold – this may need to be simulated if you do not have the data available.
2. Start JConsole and select the node *Coherence > Cache > (Cache Service Name) > (Cache Name) > 1 > back*. The last attribute *Units* will contain the size in bytes of the objects in the cache if you are using the BINARY unit calculator or the number of objects you are using the FIXED (default) unit calculator. If you are only loading 10% of the highest number of cache objects then you will need to multiply this value by 10 to get the total memory requirements for the cache objects, and then by the object size if the FIXED unit calculator is being used.

If you are indexing any object attributes then you will also need to take into account the size of these. Un-ordered cache indexes consist of the serialized attribute value and the key. Ordered indexes include additional forward and backward navigation information

Indexes are stored in memory – though they will not be counted in the cache size statistics found using JConsole. Each node will require 2 additional maps (instances of `java.util.HashMap`) for an index - one for a reverse index and one for a forward index. The reverse index size is a cardinal number for the value (size of the value domain, i.e. number of distinct values). The forward index size is of the key set size.

The extra memory cost for the `HashMap` is about 30 bytes. Extra cost for each extracted indexed value is 12 bytes (the object reference size) plus the size for the

value itself. For example, the extra size for Long value is 20 bytes (12 bytes + 8 bytes) and for a String is 12 bytes + the string length. There is also an additional reference (12 bytes) cost for indexes with a large cardinal number and a small additional cost (about 4 bytes) for sorted indexes.

With this in mind you can calculate an approximate index cost. For an indexed Long value of large cardinal it's going to be about:

Index size = forward index map + backward index map + reference + value size

Hence:

80 bytes = 30 bytes + 30 bytes + 12 bytes + 8 bytes

For an indexed String of an average length of 20 chars it's going to be about:

112 bytes = 30 bytes + 30 bytes + 12 bytes + (20 bytes * 2)

To summarize, the index cost is relatively high for small objects, but it's constant and becomes less and less expensive for larger objects.

Sizing a cache is not an exact science, as you can see. Assumptions on the size and maximum number of objects have to be made. So as an example for a Trades cache:

Estimated average size of cache objects = 1k

Estimated maximum number of cache objects 100k

5 string indexes, $5 * 112 * 100k = 56MB$

Approx cache size = $1k * 100k + 56MB = \sim 156MB$

Because each node needs to hold backup as well as primary data $\sim 312MB$ of data will actually need to be stored.

Each JVM will store on-heap data itself and require some free space to process data in. With a 1GB heap this will be approximately 300MB or more. The JVM process address space for the JVM – outside of the heap is also approximately 200MB.

Hence to store 312MB of data will require the following memory for each node in a 2 node JVM cluster:

$312MB$ (for data) + $300MB$ (working JVM heap) + $200MB$ (JVM executable) = $812MB$ (of physical memory)

Note that this is the minimum heap space that is required. It is prudent to add additional space, to take account of any inaccuracies in your estimates, say 10%, and for growth (if this is anticipated).

There are a number of techniques to reduce the size of a cache. One mentioned already is to not use backups. If this is not an option, because the cache data is being updated, then an alternative approach could be to backup up the changes, but only while they are being persisted to some external storage – where they can easily be retrieved. This can be accomplished by specifying the [backup-count-after-writebehind](#) element in the cache configuration. Another is to [intern\(\)](#) Java Strings. This may reduce the memory your data takes up if you have a significant amount of repeating String data, e.g. currency codes, but there will be a small overhead in removing the redundancy. POJ as a serialization mechanism will also significantly reduce the size of your cached objects, see above for more details. Finally, if data volumes increase, additional capacity can grow dynamically by adding additional servers with no service interruption.

In summary the memory requirements for a cache are:

$$\text{Cache Memory Requirement} = ((\text{Size of cache entries} + \text{Size of indexes}) * 2 \text{ (for backups and primary's)}) + \text{JVM working memory} (\sim 30\% \text{ of } 1\text{GB JVM})$$

How do you limit the size of your cache?

A cache can be size limited so that it never exceeds a specified number of objects or size. This allows data to be cached even when the available hardware cannot accommodate a full data set. When the cache limit is reached an eviction policy is invoked to remove old data to make way for new data. The eviction policy used can be based upon a ‘Least Recently Used’ (LRU), some other in-built policy or a custom one.

A cache with a size limit and eviction policy can read objects not in a cache ‘on-demand’ by configuring the cache to read-through the missing data from the underlying cache store, like a database or file system. The Coherence read-through mechanism will only automatically load missing objects when a key based lookup is made. It will not load missing objects when a query is performed and some objects that would match the query are not present in the cache – Coherence cannot translate object queries into SQL or file based search queries, etc. Therefore, it does not make much sense to use a size limited cache with an eviction/expiry policy when queries need to be performed against the cache, as the queries will not necessarily return the correct results.

A [size limit](#) and cache [eviction policy](#) for a cache is set by modifying the cache configuration file. Below is an example of how to size limit the number of entries a particular cache node will store:

```
...
<unit-calculator>BINARY</unit-calculator>
<!-- 100000000 = ~100 MB -->
```

<high-units>100000000</high-units>

...

Note: A `UnitCalculator` implementation gives the size of a cache node based upon the number of entries. This is the default calculator. If the `BINARY` mode is specified the [BinaryMemoryCalculator](#) is used and the physical memory (in bytes) of a nodes entries is returned, e.g. to a `JMX` bean. However, the [BinaryMemoryCalculator](#) implementation can only determine an accurate entry size if both the entry key and value are Binary objects; otherwise, an exception will be thrown during the unit calculation. The size limit is specified in the Coherence configuration file for the `high-units` for a node. In Coherence 3.4 the binary calculator is now the default.

These settings restrict the total amount of data that can be held in memory within this cache server process i.e. ~100MB (this size is in bytes). By setting cache size limits, out of memory errors can be avoided. The default eviction policy is a hybrid eviction policy that chooses which entries to evict based the combination (weighted score) of how often and recently they were accessed, evicting those that are accessed least frequently and were not accessed for the longest period first.

How many JVM's?

Generally speaking its better to use lots of JVM's with a small heap, say 512MB - 1GB, than a fewer number of JVM's with a larger heap. If the heap size is greater than 1GB then you will generally need to consider tuning the Garbage Collection (the process of recovering unused memory) settings to prevent prolonged GC times from a full GC. Running with a small heap (and setting the min and max values to the same value) means that GC times will be short and the overall performance of the data grid more consistent and responsive. Furthermore, with lots of smaller JVM's any node failure in the cluster will also have less impact on the overall cluster as less data will have to be re-balanced.

The overhead of incremental heap expansion can be eliminated by explicitly setting the minimum (`-Xms`) and maximum (`-Xmx`) JVM heap size to the same value at startup. This value used should also be used for all cache nodes in the cluster, so that all cluster members have the same JVM heap size.

The optimum memory to CPU ratio (MEM:CPU) will depend on the type of processing carried out in data grid. For simple data operations, gets and puts, then a memory to CPU ratio of 8:1 may be sufficient, e.g. 2 CPU, dual core server with 32GB of memory. However, for more CPU intensive operations, like parallel aggregations, queries, extensive use of entry processors, etc., a higher ratio of perhaps 4:1 may be required, e.g. 2 CPU, dual core server with 16GB of memory. The overhead of the operating system on a server should also be considered when determining the number of JVM cluster nodes to start, as does the management overhead. As a guide, a 2 CPU dual core server with 16GB of memory should be able to comfortably run approx. 8 to 10 JVM's, each with 1GB heaps, leaving the OS with reasonable 'head-room'. To determine the maximum number you can use `top` on Unix or the System Monitor on Windows. Although these guidelines

provide a good starting point for hardware selection, the optimum configuration should be validated through testing.

To maintain a highly available cluster, sufficient cache servers should be deployed so that if one fails the surviving n members will be able to hold the same amount of data that was previously stored by $n + 1$ cache nodes. So to protect against process failure, if n cache nodes are required to hold all the data $n + 1$ nodes must be deployed. To protect against server failure $s + 1$ servers must be deployed, where s is the number of servers required to hold all the data. To summarize:

$n + 1$ JVM's required to store all the data and protect against process failure

$s + 1$ servers required to run $n + 1$ JVM's and protect against server failure

Hence: $n + 1 / s$ JVM's per server

Generally you need to provide more contingency than 1 extra JVM or even 1 additional server, so that if a failure occurs you are not vulnerable if a further failure occurs.

A standard cache configuration and Coherence override file should be created. Multiple versions can be created for different environments, development, test and production.

Configuration Management

Establishing a common configuration file, or set of files, that is accessed centrally has been found to be the simplest way to provide a consistent cluster configuration. However, if a client application is starting up and shutting down very quickly – because for instance client libraries being used have memory leakage issues – then it is sometimes more performant to copy the client configuration files to a local location.

Add JMX monitoring settings to the Coherence JVM's and the `-verbosegc` setting to monitor GC's. This will enable any problems to be more easily diagnosed, the `-server` option to improve the JVM's performance and fixed minimum and maximum heap sizes to pre-allocate memory at start-up. For example:

```
-Dcom.sun.management.jmxremote
-Dtangosol.coherence.management.remote=true
-Dtangosol.coherence.management=all
-Dcom.sun.management.jmxremote.port=<JMX port>
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

And

```
-verbosegc
-server
-Xms1024M
-Xmx1024M
```

Also ensure that logging is configured correctly and that any output will be sent to the right place, e.g. a log file – including JVM output. A Coherence override file

may be a more appropriate place to capture configuration settings, like logging parameters, rather than passing them as command line arguments.

Coherence does not require license keys to deploy or run it. However, there are a number of different versions that are licensed differently. To ensure that you are using only the features for the version you have licensed (the default is Grid Edition), you can set the version in the [tangosol-coherence-override.xml](#) file as shown below.

```
<!-- Specify license and mode to use -->
<license-config>
  <edition-name
    system-property="tangosol.coherence.edition">
    EE
  </edition-name>
  <!--
    This value actually has to be overridden
    as a system property
  -->
  <license-mode
    system-property="tangosol.coherence.mode">
    dev
  </license-mode>
</license-config>
```

Security

Security is key for many Coherence deployments. Coherence comes with a variety of security features that enable access to cluster nodes to be secured, actions within a cache to be restricted (based upon the identity of a client) and communications traffic to be encrypted. Further information about how to incorporate all these features into your deployment can be found in the [online documentation](#).

Change Control

For relatively minor changes to the configuration of a cluster, such as adding a cache, can be done in a rolling fashion across a cluster. More drastic changes may require a cache restart. For instance, a minor patch can probably be applied in a rolling fashion but a major release upgrade will require considerable planning, which is outside the scope of this document.

How do I change my configuration on the fly?

By configuring a separate service for each cache it is possible to start, stop and re-configure caches independently from each other.

How do I change my cache objects in a production system?

If the format of an object is being changed in a production system, the cache that holds the object will need to be completely re-started, unless the change is backwardly compatible with the previous object format. Some customers have successfully tried online upgrades in their test environments but most re-start the cache with the new objects.

To perform an online update of an objects structure it has to be backwardly compatible and also written to cope with different versions being used in the same cache.

To accomplish this, each class has a version number and a binary array attribute to store new attribute data. Where an old and new version of a class exist and are deployed to the client and server platforms by rolling re-starts the following scenarios can occur:

- **Old client accessing old object and new client accessing new object**
In each of these cases each will be able to access all the attributes available on the client side object class.
- **Old client reading newer object**
If an old client accesses a new object, where the new version of the object has additional attributes, then when a client based on the older version of the object requests objects based upon the new format, the excess attribute values are stored in the binary store when it is de-serialized. When it is written back to the cache or saved the binary store data is written back, or serialized, after the old attributes. This means they are not lost.
- **New client accessing old object**
If a new client accesses an old object then the old object will be converted into a new object and the fields where there are no values to de-serialize will be given default values.

While versioning cache objects in this manner requires a small amount of extra work it makes them upgradeable in a safe manner. For more information about versionable objects see the Coherence Java interface [EvolvablePortableObject](#) and Coherence .NET interface [IEvolvablePortableObject](#).

If the new object is not compatible with the previous version then the upgrade path is more complicated and will depend on the systems architecture and other factors. There are a number of different approaches. One is to clear the old cache (which can be done from JConsole), the new classes added to the cluster nodes CLASSPATH's and each node restarted in a rolling fashion. Alternatively, a new cache can be setup in parallel and the objects gradually migrated from the old cache to the new cache and the values of the object converted in the replication. A rolling upgrade of the nodes CLASSPATH would be required and at some point a client 'black-out', to ensure all data had been migrated. During this client 'black-out' period the clients CLASSPATH could also be updated with the new classes and their configuration changed to point at the new cache.

How do I keep my .NET, C++ and Java code synchronized?

When developing applications that have .NET or C++ clients a corresponding Java object will be required, for example if queries and entry processors are going to be

run against the data stored in the cache. At the moment this requires manual coding of both the client and the server objects in their corresponding languages.

Testing should be targeted at the key application requirements of Coherence. It

should also be performed in an environment, which is very similar to – albeit smaller than – production.

TESTING

Testing Strategies

Run the basic tests tools provided by Coherence first, i.e. the [datagram test](#), the [multicast test](#) and perhaps the [access test](#). These will very quickly give you an idea of the practical limits of your test environment. For instance the datagram test will tell you how much data can be sent across your network infrastructure. So for Gbit Ethernet a maximum of 100 Mbit p/s (~10MB) can usually be transmitted. If each object is on average 10k then the maximum number of get requests will be ~10k.

These kinds of tests will tell you what is theoretically possible with your test environment before running any of your application test scenarios.

What to test?

Most systems where Coherence is being introduced already work – just not well enough or not for much longer. So Coherence is being utilized to:

- Make a system more scalable
- Make it more resilient
- Improve performance, e.g. lower latency
- Alleviate load on an existing resource through caching

Since testing is an open-ended activity, care needs to be taken to focus on those areas where Coherence is going to bring the most value.

What to measure?

The short answer here is everything. So monitor the network (which will often be a bottleneck), the OS (CPU, memory, etc.), the JVM (GC's), information/statistics from Coherence (through JMX) and test harness (number of users, messages, etc.). This will provide a picture of the cause of any bottleneck or limiting factor and allow tuning and trouble shooting to take place.

Before testing also decide on targets, SLA's, etc. that the production system will need to meet. Once they have been met, e.g. a particular response time, then focus on another system characteristic, for instance number of users.

How to test?

Try and use as representative a system as possible, making the network and server hardware as close as possible to the production environment.

Choose the difficult tests first, i.e. the hardest scenarios you expect Coherence to have to handle. Then test the system until it breaks, either because there is insufficient network bandwidth, processing capacity, cache storage, etc.

Example Proof of Concept (PoC) Tests

Measuring Latency and Throughput

When evaluating performance you try to establish two things, latency and throughput. A simple performance analysis test may simply try performing a series of timed cache accesses in a tight loop. While these tests may accurately measure latency, in order to measure maximum throughput on a distributed cache a test must make use of multiple threads concurrently accessing the cache, and potentially multiple test clients. In a single threaded test the client thread will naturally spend the majority of the time simply waiting on the network. By running multiple clients/threads, you can more efficiently make use of your available processing power by issuing a number of requests in parallel. The use of batching operations can also be used to increase the data density of each operation. As you add threads you should see that the throughput continues to increase until you've maxed out the CPU or network, while the overall latency remains constant for the same period.

Scalability

To show true linear scalability as you increase the cluster size, you need to be prepared to add hardware, and not simply JVM's to the cluster. Adding JVM's to a single machine will scale only up to the point where the CPU or network is fully utilized.

Plan on testing with clusters with more than just two cache servers (storage enabled nodes). The jump from one to two cache servers will not show the same scalability as from two to four. The reason for this is because by default Coherence will maintain one backup copy of each piece of data written into the cache. The process of maintaining backups only begins once there are two storage-enabled nodes in the cluster (there must have a place to put the backup). Thus when you move from one to two, the amount of work involved in a mutating operation such as `cache.put()` actually doubles, but beyond that the amount of work stays fixed, and will be evenly distributed across the nodes.

Data Reliability

Ensuring that data remains accurate and available even in the event of server failure is a fundamental requirement of a distributed caching technology. Data stored in the distributed caching system must be equally resilient to individual JVM failures as well as server failures (i.e. NIC failure, Ethernet cable removal, power supply failure, etc.). Further, all data access requests (i.e. access/update) must complete, all transactions must complete and the system must simultaneously load balance back to a steady state of primary and back-ups across the distributed environment.

Destructive Testing

In order to understand what a system is capable of, it is equally important to understand when it will break. If the system breaks, that failure should be

predictable in nature and easily identifiable. Determining the breaking point should be done by:

- Overloading the distributed caching technology with a larger data set than the memory that is available to see how it reacts, compensates, etc.
- Taking a large (in terms of number of objects) distributed cache and steadily increase the number of clients continuously accessing the data to see (1) what performance outliers exist and (2) if and when the system fails.
- Performing a long running test (days) in which a number of clients are accessing the data stored in a large (in terms of number of objects) distributed cache and capturing the performance characteristics to determine what performance outliers exist.

Planning and targeted testing are key to the success of this deployment phase.

DEPLOYING TO PRODUCTION

Transitioning from Test to Production

Ensure that you have followed the Production Checklist and other key documents (mentioned above). Additionally, the following have been found to ease the transition:

- Establish a standard configuration files and override files (used for common settings)
- Create a build checklist, or server image – to ensure all servers are identical, which will ease with trouble shooting

If you are using Extend clients consider using separate JVM's for the proxy servers. This is because the JVM GC parameters may need to be optimized differently as they will be creating and destroying objects at a different rate to standard cache cluster members. This will also enable the number of proxy servers to be varied independently of the cache servers.

Roadmap

An example transition roadmap for moving from a PoC to production could look something like this:

1. **Determine approximate Data Grid size** based upon application sizing tests. This can be done in a development environment providing that the production OS, etc. is the same, e.g. development OS is 32 Bit and production OS is 32 Bit.
2. Use the Production Checklist and Performance Guide to help with your **choice of hardware.**

3. **Create a representative test environment** – based on production hardware, network configuration, OS, etc. This should be as close as possible, perhaps representing a subset of the production environment.
4. Follow the Production Checklist and **tune OS, network configurations, etc.** as per Performance Tuning Guide.
5. **Perform data-gram and multi-cast tests** (assuming multi-cast is going to be permitted in the production environment, otherwise setup Well Known Addresses). This should stress your network to determine the physical limits for moving data, etc. Before you perform the data-gram test ensure that the network segment is isolated from rest of organizations network – it will generate a lot of traffic.
6. **Configure the data grid for production** - e.g. set mode to 'prod', add JMX parameters, etc. and place configuration files in a central location, e.g. a shared drive or Web Server (URL's can be used to reference the cache configuration file).
7. **Deploy your application using chosen production management tools**, starting/stopping the data grid (e.g. via WebLogic Operational Control, scripts) to check that it works correctly. Also install any monitoring tools, like WebLogic Operational Control, and ensure that the required operational management information is being captured correctly.
8. **Re-check sizing calculations** in test environment.
9. Once the test environment has been setup as per production then **perform targeted tests on test platform**. See the Test Strategies section above for more details.
10. **Iteratively tune and re-test the application** until you are getting the required level of performance and have exhausted all available options – in terms of Coherence, network, OS, JVM tuning, etc.
11. **Replicate and scale-up the test environment in the production environment.**
12. **Re-test** to ensure sufficient capacity and comparative performance. Check monitoring and management tools are functioning correctly and then deploy.

This document should help you plan your Coherence application deployment. However, to get the most from the Coherence this white paper should be read in conjunction with the other product guides.

CONCLUSION

It is very important to thoroughly read the guideline documents for Coherence in conjunction with this document, to gain an overall picture of the steps required to move a Coherence environment from a PoC to production. Ultimately every application uses Coherence in a slightly different manner but the overall principles outlined above should hold true. Planning and targeted testing in a representative environment should minimize the risks involved in a new Coherence deployment.



Coherence Planning: From Proof of Concept to Production

November 2008

Author: Dave Felcey

Contributing Authors: Jon Purdy, Rob Misek, Gene Gleyzer, Jon Hall, Brian Oliver, Peter Utschneider, Michele DiSerio and others

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Copyright © 2008, Oracle. All rights reserved.

This document is provided for information purposes only and the contents hereof are subject to change without notice.

This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission. Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.