Detecting and Alerting on Fault Conditions in an Oracle Coherence Distributed Caching System

Thomas Lubinski
SL Corporation
Corte Madera, CA
February 15, 2011

***Abstract* – Oracle Coherence is a powerful, yet complex, distributing caching system used in many large mission-critical applications. Fault conditions such as lack of available memory or excessive network communication can lead to intolerable data loss or even complete cluster failure. Detecting these dangerous conditions using intelligent alerts as early as possible is critical to preventing outages and ensuring high availability in such systems. Techniques for achieving this using available JMX monitoring data are presented in the context of commonly encountered use cases.**

I. Introduction

Oracle Coherence is an in-memory distributed data grid and caching solution for applications and application servers. Many large business applications – in industries as diverse as financial services, risk management and on-line stores – use Coherence services for storing and efficiently accessing large volumes of data. However, as a powerful and complex distributed system, it must be managed effectively in order ensure its uptime and performance in critical applications.

The author of this article and SL Corporation have over 25 years of experience with monitoring and visualization applications, with particular expertise in Java. The company's RTView Oracle Coherence Monitor product has been specially adapted to deal with the large volumes of real-time data produced by the JMX monitoring MBeans in Oracle Coherence, and has features designed to generate intelligent alerts based on regular monitoring of these metrics.

Several examples are presented in this paper to illustrate how timely alerts can be distilled from the mass of data produced in a large operational cluster. Several commonly seen use cases are described to highlight the important metrics and how they change as cluster operations result in conditions that are unhealthy for the cluster.

This is not intended to be a definitive list of all possible patterns; rather, it simply suggests an effective methodology that can be used in addressing requirements for complex monitoring applications.

II. The Coherence Vision

A finely-tuned and smoothly operating Coherence cluster is a wonder to behold. Dozens of networked computers running hundreds of individual Coherence nodes are linked together by an advanced internal network protocol to create a huge high-performance in-memory virtual data storage bank, with capacity seen as high as 500 gigabytes or more. Data that are not contained in the fast caches are transparently swapped in from even larger databases operating behind the scenes.

From the point of view of an application programmer, data in this super cache are accessed using simple "get" and "put" operations as if the cache were running on the user's single machine. This eliminates any need to worry about the location of the data, or whether it is in a database or not. The data are just "there" and readily available. Additionally, Coherence provides a built-in backup mechanism ensuring data integrity even if a few machines in the cluster go down or become unavailable for a period of time.

This is a highly valuable concept, and has found use, for example, in many large applications in financial services where large amounts of trading information are processed to keep a real-time view of current positions. Or, a large on-line retail operation may use caches to manage product inventory and availability in response to user purchases. Applications such as these can be written and maintained much more easily than before because of the much simpler data model.

For application programmers with access to a Coherence data store, life is made much easier. The burden of managing the location of the data is the responsibility of the Coherence infrastructure, not the programmer as has been the case for years.

Coherence accomplishes this with software that integrates many individual "nodes" or JVMs, each of which manages a small subset of the data stored in the large virtual cache, along with backup copies of the data. Clusters can range from just a few nodes up to many hundreds in some larger clusters. Each node communicates with every other node in the cluster and

complex algorithms are used to locate each piece of data in the cluster, or on persistent backup storage such as a database. This is all transparent to the user.

Figure 1 – A Coherence cluster consisting of many nodes

To the user, a Coherence cluster is like a "black box." There is no GUI or central console to show you the cluster configuration, the data or the current state. You put data in and take it out, but what goes on behind the scenes is very complex. When it works, it is beautiful. However, when something goes wrong, it can be quite difficult to determine the source of problem and what to do to fix it. In fact, if something goes wrong in your application, it can be difficult to determine if the cluster is at fault at all.

III. Monitoring the Cluster

Monitoring a Coherence cluster in real time is important to ensuring its uptime, performance and reliability. As one Coherence power user noted, "Coherence is a high performance system. That means when it goes south, it goes south quickly. If it runs out of memory, all the data is gone."

While many infrastructure monitoring systems perform a health check every five minutes, most users agree that measurements in Coherence should be made more frequently in order to have sufficient warning to correct problems before they result in an outage. At SL, we usually suggest that monitoring measurements be taken every 10-30 seconds.

Coherence exposes a great deal of detailed information about its internal operation which can be used for alerting, troubleshooting, performance analysis and capacity planning. However, by its nature, monitoring Coherence can be quite complex. A cluster containing 100 nodes and managing 20 uniquely named data stores, or caches, will have at a minimum 20 x 100, or 2,000 individual test points that must be monitored. In fact, there are actually more like 10,000 test points in a moderately sized cluster. Making sense of all this data is a huge undertaking in itself.

In a previous paper, the author presented in-depth the JMX monitoring MBeans that were available from Coherence, and some techniques for optimizing the retrieval of this large amount of data from a Coherence cluster. Additionally, this paper discussed some of the information that was missing from the Coherence MBeans and ways to augment the monitoring using special configurations and other approaches.

Many developers are familiar with using JConsole or other simple JMX management tools to look at individual MBeans, or to perform some simple aggregations. However, making sense of thousands of MBeans at once requires special handling and products specifically designed to aggregate and organize large amounts of data within context. For example, to get a view of the amount of data going into and out of a single cache in the cluster, MBean data from each of the nodes in the cluster must be summed. One node by itself provides only a very limited view. Additionally, there are often calculations and deltas that must be performed on the various data tables in order to extract useful information.

Figure 2 shows a breakdown of MBeans that are available in a small cluster with 8 storage nodes running a couple of services and about a dozen caches.

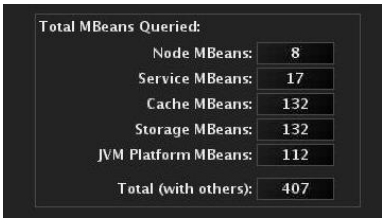

Figure 2 – Sample cluster MBeans by type in a small cluster

It is not practical to monitor or even troubleshoot the cluster above using JConsole to examine one MBean at a time. There needs to be an automated approach.

IV. Detecting and Alerting on Problems

Monitoring a Coherence cluster properly can provide early warning of trouble. If one can quickly detect that memory is running low, for example, additional storage nodes could be allocated, increasing the total capacity before the system runs out of memory. Monitoring systems based on retrieving and analyzing data in a relational database typically have too much latency to be effective.

The following is quick summary of metrics that are useful for detecting problems in Coherence.

Table 1 – Some MBeans available from Coherence

MBean	Relevant Information
Node	Location, CPU/Memory Usage, Network Failures
Service	CPU Usage, Thread Usage, Task Backlog, H/A Status
Cache	Object Count, Memory Size, Total Gets / Hits / Misses
Storage	Evictions, Index Data
Platform	JVM Memory Usage with Garbage Collection detail

The sections that follow discuss ways that these metrics can be used to proactively detect and alert on unhealthy conditions.

A. *Endangered Data – Node and Machine Safety*

Coherence provides one very important piece of data for every service running in the cluster. This is the H/A Status (for high-availability). This very simply indicates whether the data in the caches running on a specific service are "SAFE" from being lost if either a node or machine goes down.

Cache Services

Service Name	StatusHA	Total Nodes	Storage Nodes	Caches	Objects	Senior
DistributedCache	NODE-SAFE	27	6	4	1,034,165	41
DistributedCache-A	MACHINE-SAFE	29	11	5	10,431	42
MyDistributedCache	MACHINE-SAFE	19	7	4	7,456	44
OnLineStoreCache	MACHINE-SAFE	4	2	2	34,942	63

Figure 3 – Simple table showing Cache Services status

The table above is from a monitoring display showing summary information about each service in a cluster. The name of each service is shown, along with its H/A Status, and other information about the service such as number of objects, number of nodes running the service, and number of individual caches.

Note that the second service is MACHINE-SAFE. This means that an entire machine could go down, but data in the caches on that service may not be lost because there are enough storage nodes so that backups can be distributed on the reduced number of machines. However, MACHINE-SAFE status is NOT an assurance that the *cluster* will have enough memory to survive a machine failure. After a failure, primary data, backup data, and index information will need to be redistributed to the remaining machines. These machines need to have enough available memory to store the new data.

The first service however, is only NODE-SAFE. This means that the data are safe if a single node goes down, but if an entire machine goes down, data will likely be lost. This NODE-SAFE condition may be temporary because data redistribution has not completed after a previous event. Or it could be that there are not enough memory and machines to distribute all primary and backup data on separate machines. If a machine goes down, some data will likely be lost. A service can also be ENDANGERED meaning that if any node goes down, data will likely be lost, as the backup data cannot be distributed effectively enough to ensure otherwise.

Since Coherence provides this information in a standard MBean field, it is easy to detect one of these situations, and inform a user that a problem has occurred. Typically, an alert is generated, resulting in an email or text message to an operator who could take corrective action. Sometimes an SNMP trap is used to pass the alert to an in-house system management console.

B. *Departed Node – Node Leaving Cluster*

In a large cluster, data may be safe even if more than a single node goes down. This is because there is enough back storage available on other nodes to distribute the load.

However, if any node goes down, it is usually indicative of something starting to go wrong that may need to be addressed immediately. For example, one node might go down due to network problems or running out of memory. This could tax the cluster, leading to a chain reaction where additional nodes get overloaded and begin to go down resulting in catastrophic failure. Detecting that the first node is down could get an operator to start investigating the situation and possibly take action that could prevent disaster.

Through version 3.5, Coherence does not provide any automatic indication that a node has left the cluster (future releases provide this as a JMX notification). The only way to determine that a node has died is to regularly query all of the Node MBeans from all nodes in the cluster, and compare the list of MBeans returned with the list obtained in the most recent query. If a node is missing, one can conclude that it must have died during the previous interval.

There is a refinement to this alert that may be necessary. The safety of data is dependent only on storage-enabled nodes, so the alert should only be generated if one of these goes down. In some applications, client or process nodes can come and go at different times and an alert on the departure of one of these is not useful. Thus, flexibility in the definition of the alert is needed in a general-purpose monitoring system.

Detecting that a node has gone down is useful, but often even this is too late. Ways to detect conditions that lead up to nodes dying are discussed below.

C. *Monitoring Memory Utilization*

At a basic level, a Coherence cluster can be thought of as one super data store. The capacity of that store is limited by the amount of heap memory allocated across all the nodes in the cluster. If objects are arbitrarily added to the cluster without monitoring the utilization of that available memory, the cluster can fill up resulting in fatal OutOfMemory errors on some of the nodes, quickly cascading to complete cluster collapse.

As a general rule, detecting problems in total cluster memory utilization is a last line of defense. Section D below will discuss ways to more granularly control capacity on individually named caches.

A commonly seen type of monitoring display makes use of a heatmap to show memory utilization within a cluster. In the display below, each node is represented as a rectangle where the size or area of the rectangle represents the heap memory allocated to the node, and the color represents the percent of the heap memory currently in use by the node.

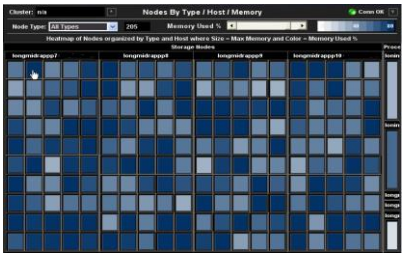

Figure 4 – Heatmap showing memory utilization of all nodes

Note that in this diagram, the process nodes are shown in a separate grouping from the storage nodes. As with node death, there is often a difference in the rules that apply to process nodes and to storage nodes.

Monitoring node memory utilization at this level is not as easy as it first looks. You cannot just assume that the memory reported by the JVM is accurate. The task is complicated by the effect of Java's built-in garbage collection mechanism.

Two issues cloud the picture. First, the memory reported as "used" by the JVM includes objects that have been deleted but not yet removed (referred to as garbage), making it difficult to tell exactly how much memory is really in use. Second, when the JVM decides to perform a garbage collection operation, all processing in that node stops temporarily and can result in long delays in communication with other nodes.

The trend chart below shows the memory utilization reported by a sample node. The sawtooth shape of the trace highlights the effect of garbage collection in this node.

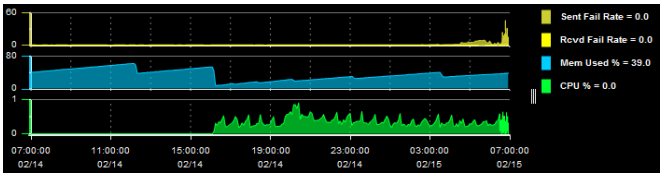

Figure 5 – Chart showing single node memory utilization (blue)

Over a period of time, the reported memory utilization increases regularly as objects are being added or removed in this node. Note however, that after each garbage collection event, the reported utilization drops down to about the same level as before. This indicates that memory utilization is essentially level and not fluctuating as much as reported.

The memory utilization heatmap shown in the earlier figure thus is highly misleading. It appears that the nodes are varying widely in their utilization. The reported memory depends entirely on where in the sawtooth the query was made.

This makes it difficult to set a level on which to generate an alert. In many cases, the JVM can let memory grow to a very high percentage of what is allocated before performing garbage collection. The only way to know for sure that there is a problem is by using the level at the bottom of the sawtooth as a trigger for the alert. When the bottom of the sawtooth reaches a specified level, an alert could be generated, letting users know of a growing problem in the cluster.

The JVM Platform Beans provide detailed information about the behavior of the garbage collector in Java. Another useful piece of information is the length of time that it takes to perform a garbage collection. A very long garbage collection timeframe can also be an indicator of another sort of problem. If a cache consists of many small objects and the nodes are allocated with large heap space, garbage collection can take a very long time, resulting in communication delays. This is another metric that can be used to generate an alert.

As a last line of defense, detecting and alerting on low memory can be useful. However, it doesn't provide a clue as to where the memory *is* being used. Fortunately, Coherence provides finer control over the individual caches of data that are managed in a cluster.

D. *Monitoring and Controlling Cache Capacity*

Each cache in a cluster can be configured to have a limit on how much data it can hold. Any attempt to put additional objects into the cache will result in older objects being evicted.

This feature has a dual purpose. One, it puts a limit on the amount of memory that can be utilized by a specific group of objects. If all caches are configured in this way, then the maximum utilization of memory can be controlled ahead of time. Two, it also provides a mechanism that supports quick access to frequently accessed data; if an object that is queried is not in the cache, it is retrieved from a database and replaces other objects in the cache.

The Cache MBeans return both current memory used in a cache along with the memory limit. The metric is referred to as "Units" because it can be configured to report in either number of objects or in bytes. It is most useful when set to bytes, as it represents the amount of memory consumed.

In the display below, a bar chart shows the amount of memory used (Units) in a number of caches, along with the High Units, or maximum. Some caches are near their capacity; others not so.

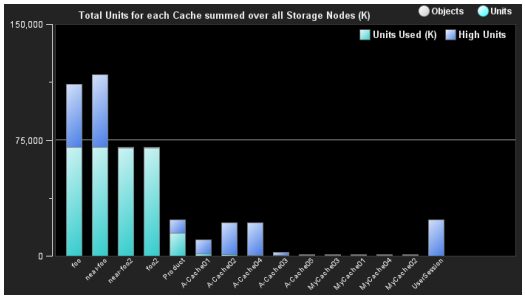

Figure 6 – Bar chart showing Cache Units and High Units

By taking advantage of the High Units setting, the available JVM heap space on each node can be segmented and carefully controlled. Each cache can have a specific size limit, which can then be monitored. Of course, not all applications have use cases appropriate for this feature. Sometimes a combination is useful. Some caches can be left to freely grow.

A typical use is where a cache is acting as a buffer against a database. In this case it would be desired to allocate a certain amount of space to act as a buffer. Once the space is filled up, objects may be evicted. It might be desirable to monitor and alert if too many evictions are occurring.

The heatmap below is an example of the how High Units effectively partitions the available memory on a node.

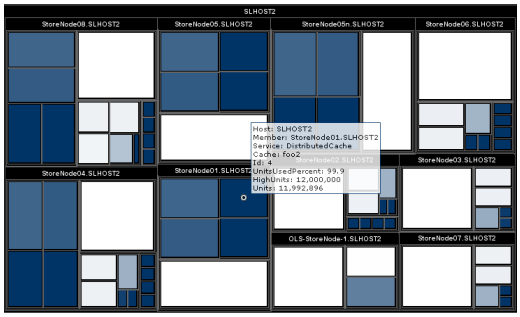

Figure 7 – Heatmap showing partitioning of caches on nodes

In this heatmap, each labeled node holds data for about a dozen caches. The size of the box represents total capacity for the cache, and the color indicates what percentage of that capacity has been filled (dark blue means 100%).

E. *Cluster Communication Failures*

Coherence cluster members communicate through Coherence TCMP, a UDP-based protocol. TCMP originally stood for Tangosol Communication Management Protocol. It combines the speed of UPD with the reliability of TCP.

Cluster node communication failures are a symptom in nearly every cluster degradation situation that we have encountered. For this reason, the TCMP publisher/receiver failures are one of the most important metrics to track.

JMX provides the publisher/receiver success rate data per cluster node, but this metric should be ignored because it's provided as an average since the cluster node started. It won't tell you if failures are occurring right now. The better way is to calculate this metric in the monitoring system (as in RTView), taking deltas from the total packets sent and received over the last collection interval and then calculating percentages. In degradation situations, high failure rates are often seen, up to 40 and 50% packet failures.

What causes these failures? Interestingly, such failures rarely result from network problems. Since multicast is used, a lot of information is broadcast between the nodes. A common scenario is when a storage node leaves the cluster. In this case, the remaining nodes communicate with one another to discover that a node has become unresponsive and must be declared "dead." Once it has, the backup data stored on other nodes must then be changed to primary data; backup data must be recreated and distributed among the other cluster nodes. Several rounds of data redistribution can ensue. Typically, this results in a short storm of network activity. As nodes are busy processing the data, the packet publishers and receivers time out and the result is a report of packet retransmissions (the result of packet failure). If the original node then rejoins the cluster, another round of data distribution is triggered. The Quorum policy in Coherence 3.6 allows users to configure clusters so as to minimize this redistribution.

Usually, when a node leaves the cluster, there is a short burst of packet failures, lasting at most a few seconds while the data are redistributed. Any alert defined against this data must account for the short burst and not trigger a major alarm, as the condition is usually temporary and the result of a node going down for maintenance, as is common. In the same way, it is normal to see communication failures during cluster startup and during cache warm-up processes.

However, there is another case that is more important to catch. This relates to garbage collection.

Depending on the pattern of activity in the cluster, garbage collection may become a problem. If data are being put into and removed from a cache quickly, garbage can be accumulating. Other cluster operations will also produce garbage such as entry processors, building and updating indexes, write-behind queues, etc.

The JVM must respond to this by periodically collecting the garbage and removing it. This process can be very time consuming. A lot of technology has gone into optimizing this and providing options that behave differently for different use cases.

However, when garbage collection is occurring, all other activity in the nodes stops. This means that packet transfers are not processed in a timely manner, resulting in a significant increase in communication failures.

Thus, monitoring of communication failures is a critical step in preventing cluster failures. Typically, the pattern is that communication failures are seen, followed by an increase in post-GC memory levels because the garbage collector is not doing an adequate job and garbage is not being removed. This eventually results in the death of that node when it runs out of memory. The result of this is a redistribution of data, meaning even more packet loss and delays, with a result that the entire cluster comes down. Again, the quorum policy in Coherence 3.6 is designed to reduce this churn and also allow the cluster to freeze rather than continue in a downward spiral.

Effectively monitoring for communication failures means that there needs to be a setable duration, so that transient innocuous redistributions do not generate an alert. A longer period of communication failures above a certain level is a sure indication of a problem brewing in the cluster.

Shown below is a typical communication pattern seen when garbage collection is causing too many packet failures. In this case, a trigger level could be set to generate an alert which could then be investigated and corrected.

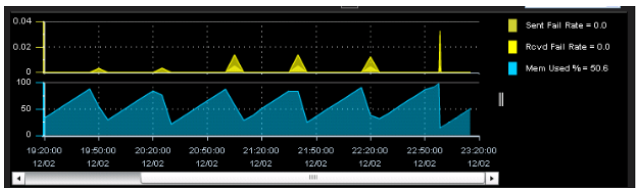

Figure 8 – Garbage collection (blue) causing packet loss spikes

Monitoring of communication failures can provides a warning, but by the time communication failures have gone

up, it may already be negatively affecting cluster performance. Another way to detect this type of problem is to monitor the data access in the cluster by observing the distribution of requests across the storage and processing nodes.

F. *Hot Keys, Excessive Requests on One Node*

An interesting case commonly seen revolves around the problem of "hot keys." Coherence is very good at dealing with data access patterns where data requests are distributed (for the most part) evenly across a data set. Sometimes however, the nature of the data means that certain objects may be accessed more often than others in a particular time interval.

Access to the same data over and over again means that the one node that is holding the official copy of the object is going to be accessed repeatedly. The node will only be able to service one request at a time (more if additional threads are allocated, but it is still limited). This means that nodes will have to wait for their requests to be serviced. If a node is perceived as being non-responsive for too long a period of time, it may be declared "dead" by other members of the cluster. The node is then kicked out of the cluster.

Once a node is ejected, data redistribution occurs, but the new node that contains the data has to start responding to the requests and eventually it may be kicked out as well.

The chart below shows a history of the CPU level on all process nodes (top) and storage nodes (bottom) in a cluster. About halfway through the period, one node starts to show a high number of requests and thus high CPU levels. This is due to the fact that all the process nodes are making requests for the same piece of data which is stored on the one node.

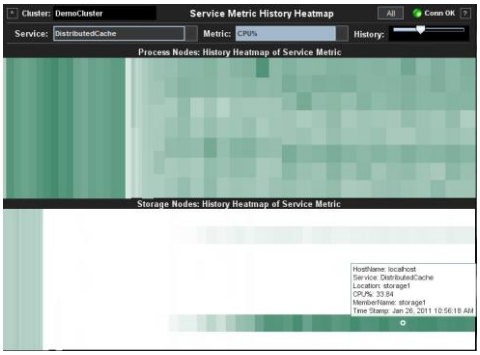

Figure 9 – History of CPU level for process and storage nodes

Often in such a scenario there will be a number of other indicators besides CPU level. For example, network communication failure rate will typically rise. If this cluster were more heavily loaded, it is likely that the node would eventually be ejected from the cluster.

Coherence unfortunately does not provide any way to determine the number of requests on a specific key. Of course, an application can be instrumented with JMX to provide this type of information. While it is a good practice that monitoring be designed into applications, it is unfortunately usually an afterthought.

V. SUMMARY

It is important in Coherence to monitor and understand chronic conditions that threaten the integrity of Coherence's clustering protocol and threaten the integrity of the cluster itself. Cluster failure can be catastrophic because failure of an in-memory system results in loss of all data. As one user observed, it is like a car crash without any damaged cars left at the scene of the accident to understand what happened.

By monitoring critical operational metrics like TCMP publisher/receiver failures, garage collection pause time, and post GC available memory, most impending failures may be avoided. Also by collecting and persisting monitoring metrics, data will be available for post mortem analysis if something should go wrong.

REFERENCES

[1] Oracle – Oracle Coherence Knowledge Base Home, October 2010, http://wiki.tangosol.com/display/COH/Oracle+Coherence+Knowledge+Base+Home

[2] Oracle - Coherence Planning: From Proof of Concept to Production, An Oracle White Paper, November 2008

[3] Oracle - Coherence 3.6 Documentation, Oracle Technology Network, http://www.oracle.com/technetwork/middleware/coherence/documentation/index.html

[4] Lubinski, Thomas – Monitoring Oracle Coherence using JMX: Challenges and Limitations, Technical Papers, www.sl.com, SL Corporation, 7 October 2009

[5] Lubinski, Thomas – Practical Considerations When Instrumenting Applications with JMX, Information Week, July 2008