技術論文

Oracle Coherence
分散キャッシング・システムの
障害状況の検知とアラート

トーマス・ルビンスキ
SL Corporation
カリフォルニア州コルテ・マデラ市
2011 年 2 月 15 日

要約 **– Oracle Coherence は強力でありながら複雑な分散キャッシング・システムであり、多くの大規模なミッション・クリティカル・アプリケーションで使われています。利用可能なメモリがない、あるいはネットワーク通信が過剰などの障害状況により、容認できないデータ喪失や、さらには完全なクラスタ障害を招きかねません。これらの危険的状況を、インテリジェントなアラートを使ってできる限り早期に検知することは、こうしたシステムで停止を防ぎ、高可用性を保証するためには欠かせません。JMX から得られる監視データを使ってこれを実現する技術を、よく遭遇する実際の使用例を取り上げながら解説いたします。**

I. はじめに

Oracle Coherence は、アプリケーションやアプリケーション・サーバのためのインメモリ分散データ・グリッドとキャッシングのソリューションです。金融サービス、リスク管理、オンライン・ストアなどの多岐にわたる業界で多くの大規模ビジネス・アプリケーションが Coherence サービスを使い、大量のデータを格納し、効率的にアクセスしています。しかし、強力かつ複雑な分散システムのため、重要なアプリケーションでの可用時間や性能を保証するには、効果的な管理が不可欠です。

本記事の著者と SL Corporation（以下 SL 社）は、監視と可視化のアプリケーションで 25 年以上の経験があり、特に Java については専門知識を有しています。同社の RTView for APM | Oracle Coherence Monitor 製品は、Oracle Coherence の JMX 監視 MBean が生成する大量のリアルタイム・データを扱うのに特化して作られています。そして、こうした指標データの定期的な監視に基づいてインテリジェントなアラートを生成するように設計された機能があります。

本書ではいくつかの例を提示して、大規模運用クラスタで生成された大量データからタイムリーなアラートを抽出できるようにする方法を説明します。そして、よくある実際の使用例をいくつか記して、重要な指標をハイライトし、クラスタ運用がクラスタにとって不健全な状況に陥るとそれらがどう変化するのかを説明します。

本書は起こりうるあらゆるパターンの決定版リストを意図したものではなく、むしろ単に複雑な監視アプリケーションに対する要求条件に取り組む際に使える、効果的な方法論をご提案するものです。

II. Coherence のビジョン

細かく調整され、円滑に運用されている Coherence クラスタは見るのも驚異です。ネットワークに接続された何十ものコンピュータが何百もの個別の Coherence ノードを実行し、それらが先進的なネットワーク・プロトコルで結ばれて、巨大な高性能インメモリ仮想データ・ストレージ・バンクを作り上げ、その容量は 500 ギガバイト以上になることもあります。速いキャッシュに含まれていないデータは、現場の背後で運用されているさらに巨大なデータベースから透過的にスワップされて来ます。

アプリケーション・プログラマの視点からは、この超キャッシュのデータに単に "get" と "put" の操作を使ってアクセスするので、まるでキャッシュはユーザの単一マシンで実行されているかのようです。これによってデータがどこにあるか、データベースにあるのかないのか、に煩わされる必要がなくなります。データはただ「そこ」にあり、すぐに得られます。さらに Coherence は組み込みのバックアップ機構を備えているので、クラスタ内のマシンが数台ダウンしてしばらく使えなくなってもデータの保全は保証されています。

これは非常に価値ある概念で、たとえば、金融サービスにおける大量の取引情報を処理し、現在のポジションをリアルタイムで見続けられるようになっている数多くの大規模アプリケーションで、その有用性が見出されてきました。あるいは大規模なオンライン小売事業で、キャッシュを使ってユーザの購入に応じた製品の在庫と入荷時期を管理するといったこともできます。こうしたアプリケーションは、よりシンプルなデータ・モデルによって、以前よりもと簡単に作り保守していくことができます。

Coherence のデータ格納域にアクセスするアプリケーション・プログラマにとっては、さらに楽です。これまではプログラマの責任であったデータの在処を管理する重荷は、Coherence のインフラが担うからです。

Coherence は、多数の個別の「ノード」すなわち JVM を統合するソフトウェアでこれを実現し、そこでは個別のノードが、巨大な仮想キャッシュ内に格納されたデータの小さなサブセットを、そのデータのバックアップ・コピーとともに管理します。クラスタは単に数個のノードから、より大規模なクラスタ内では数百までの範囲に及びます。それぞれのノードはクラスタ内における他のすべてのノードと通信し、複雑なアルゴリズムを使って、1つ1つのデータがクラスタ内あ

るいはデータベースなどの永続性のあるバックアップ・ストレージのどこにあるかを調べます。これはすべてユーザには透過的です。

図 1 – 多数のノードを含む Coherence クラスタ

ユーザにとっては、Coherence クラスタは「ブラック・ボックス」のようなものです。クラスタの構成やデータや現在の状況を表示する GUI や中央コンソールはありません。ユーザはデータを入れ、取り出しますが、その背後で起こっていることは非常に複雑です。うまく動いているときはとてもすばらしいのですが、何かがおかしくなると、その問題の原因や、どうやって直すかを判断するのはとても難しくなります。実際、運用中のアプリケーションで何かがおかしくなった場合、クラスタのエラーなのかどうかを判断するのは難しくなります。

III. クラスタの監視

Coherence クラスタをリアルタイムで監視することは、可用時間や性能や信頼性を保証するのに重要です。ある Coherence のパワー・ユーザが、「Coherence は高性能なシステムである。すなわち反対方向へ走り出すときはたちまちにを意味し、メモリが足りなくなると、データはすべて無くなってしまう」と、注意しています。

多くのインフラ監視システムでは健全性チェックを 5 分ごとに行いますが、Coherence の場合には、ほとんどのユーザはもっと頻繁に計測を行って十分な警告を出し、停止しないうちに問題を正すべきであることに同意しています。通常 SL 社では、Coherence の監視測定値は 10 秒から 30 秒毎に取ることを勧めています。

Coherence は、自身の内部運用についての詳細な情報をかなり多く外部に公開しているので、これらをアラート、トラブル対策、性能分析、容量計画に使えます。しかし、その性質上、Coherence を監視することはとても複雑になりがちです。たとえば、100 ノードが含まれ、一意的な名前の付いた 20 のデータ格納域を管理する一つのクラスタには、少なくとも 20x100、すなわち 2000 の個別のテスト・ポイントがあり、これを監視しなくてはなりません。実際には、控えめなサイズのクラスタでも 10000 以上のテスト・ポイントがあるのが実情です。このデータすべてを理解しようとすることは、それ自体途方もないことです。

以前の文書で著者は Coherence から得られる JMX 監視 Mbean と、Coherence クラスタから来るこの大量のデータの取得を最適化するテクニックのいくつかについて詳しくご説明しました。さらに同書では Coherence MBean からは得られない情報のいくつかと、特殊な構成やその他のアプローチを使って監視を増強する方法を検討しました。

多くの開発者の方々がご存知のように、JConsole や他の簡単な JMX 管理ツールを使って個別の MBean を見たり、簡単な集約を行えます。しかし何千もの MBean を一度に解釈するには、大量のデータをコンテキストに集約整理するよう設計された特別な処理と製品が必要です。たとえば、クラスタ内の単一のキャッシュに出入りするデータ量のビューを得るには、クラスタ内の各々のノードからの MBean データを合計しなくてはなりません。1つのノードはそれだけでは限定されたビューしか提供しません。さらに、役に立つ情報を引き出すためには様々なデータ・テーブルで計算したり差分を取らなければならないことがよくあります。図 2 は、8 個のストレージ・ノードが 2 つのサービスとおよそ 1 ダースのキャッシュを実行している小さなクラスタから得られる MBean を箇条書きにしたものです。

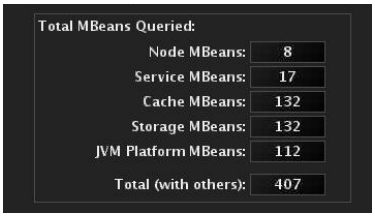

図 2 – 小さなクラスタ内のタイプ別クラスタ MBean のサンプル

このクラスタを JConsole を使って一時に1つずつ MBean を調べて監視し、あるいはさらにトラブル対策を行うのは実用的ではありません。自動化されたアプローチが必要です。

IV. 問題の検知とアラート

Coherence クラスタを適切に監視すると、トラブルを早い時期に警告できます。メモリが少なくなっていることをすばやく検知できれば、たとえば、システムのメモリがなくなる前にストレージ・ノードをさらに割り当てて総容量を増やすことができます。リレーショナル・データベースにあるデータを取り出して解析するような監視の仕方ではあまりの遅延があって、概して効果的ではありません。

以下は Coherence での問題を検知する役に立つ指標の早見表です。

表 1 – Coherence から得られる MBean のいくつか

MBean	関連する情報
Node	ロケーション、CPU/メモリ使用、ネットワーク・エラー
Service	CPU 使用、スレッド使用、タスク・バッキング、H/A ステータス
Cache	オブジェクト数、メモリ・サイズ、GET/HIT/MISS の合計
Storage	退避、インデックス・データ
Platform	JVM メモリ使用＋ガーベージ・コレクション詳細

以下のセクションでは、これらの指標を使って不健全な状況の予兆を検知し、アラート通知する方法について論じます。

A. *危険にさらされたデータ — ノードとマシンの安全性*

Coherence はクラスタで実行しているどのサービスにも1つの重要なデータを提供しています。これは（高可用性のための）H/A ステータスです。これは非常に単純に、ノードやマシンがダウンしても特定のサービスで実行しているキャッシュ内のデータが失われずに「安全」かどうかを示すものです。

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

図 3 – キャッシュのサービス・ステータスを表示する簡単なテーブル

上のテーブルはクラスタ内の各サービスについてのサマリ情報を表示する監視画面からのものです。各サービスの名前が表示され、続いてその H/A ステータス、それにオブジェクト数や、そのサービスで実行しているノード数や個別キャッシュの数など、そのサービスについての他の情報が表示されています。

注目していただきたいのは、2つ目のサービスが MACHINE-SAFE であることです。これはマシン全体がダウンしてもストレージ・ノードが十分にあって、そのサービス上のキャッシュ内のデータを失わずにいられるであろうため、より少ないマシンにバックアップを分散で可能、という意味です。しかし MACHINE-SAFE ステータスは、マシン・エラーを乗り切るだけのメモリがクラスタにあるという保証ではありません。マシン・ダウン後、プライマリ・データとバックアップ・データとインデックス情報を残りのマシンに再分散させる必要があります。残りのマシンには、この新しいデータを格納するだけの利用可能なメモリがなくてはいけません。

しかし、1番目のサービスはただの NODE-SAFE です。これは単一のノードがダウンしてもデータは安全ですが、マシン全体がダウンするとデータが失われる、ということです。この NODE-SAFE の状況は、発生したイベント後のデータの再分散がまだ完了していないための一時的なものかもしれません。もしくは、プライマリ・データとバックアップ・データを別のマシンに分散できるだけの十分なメモリやマシンがないこともあり得ます。一つのマシンがダウンすれば、恐らくいくらかのデータが失われます。そしてサービスも ENDANGERED、つまり危険にさらされる可能性があり、これは、もしノードのいずれかがダウンすれば、データが失われないと保証できるほど効果的にバックアップ・データを分散できないためデータが失われるだろう、という意味です。

Coherence はこの情報を標準の MBean フィールドで提供しているので、こうした状況のいずれかを検知し、ユーザに問題発生を知らせるのは容易です。一般的にアラートが生成されると、それを正す対応アクションを取れるオペレータに電子メールかテキスト・メッセージを送ります。またときには、SNMP トラップを使ってアラートを内部のシステム管理コンソールに通知します。

B. *離脱ノード — クラスタを離れるノード*

大規模なクラスタでは、1つ以上のノードがダウンしてもデータは安全かもしれません。これは、他のノードに利用可能なバックアップ・ストレージが十分にあって、負荷を分散できるからです。

しかし、どのノードであってもダウンするということは、何かがおかしくなり始めたことを示していることが多く、即座に対処する必要がある可能性があります。たとえば、ネットワークの問題かメモリ不足で1つのノードがダウンするかもしれません。これはクラスタに負担を与え、連鎖反応となって、次々とノードが過剰負荷となりダウンし始め、結果的に壊滅的なエラーになることもありえます。最初のノードのダウンを検知できればオペレータに状況の調査を始めさせて災害を防ぎうる対策を取れる可能性があります。

Coherence はバージョン 3.5 からノードがクラスタを離脱したことを自動的には示してくれなくなりました（将来のリリースではこれは JMX 通知として提供されます）。ノードが死んだことを判断する唯一の方法はクラスタ内の全ノードからすべての Node MBean を定期的にクエリし、返ってきた MBean の一覧を一番最近のクエリで得た一覧と比較することです。ノードがなくなっていれば、この前のインターバル内にそのノードは死んだにちがいないと結論付けられます。

このアラートには改良が必要かもしれません。データの安全性はストレージを有効にしたノードにのみ依存しますから、このアラートはそのいずれかがダウンした場合のみ生成すべきです。アプリケーションによってはクライアント・ノードすなわちプロセス・ノードは時によってやって来たりいなくなったりしますから、そのうちの1つが死んでアラートを出すのは有用ではありません。このように、一般的な目的の監視システムにはアラート定義の柔軟性が必要になります。

ノード・ダウンの検知は有用ですが、それでさえ遅すぎることがよくあります。ノードの死につながる条件を検知する方法を以下で検討します。

C. *メモリ使用状況の監視*

基本的なレベルでは、Coherence クラスタは1つの超データ格納域と考えることができます。その格納域の容量は、クラスタ内の全ノードにまたがって割り付けられたヒープ・メモリの量で制限されています。その利用可能なメモリの使用状況を監視しないでオブジェクトがクラスタに勝手に追加されると、クラスタがいっぱいになります。そして、その結果ノードのいくつかで致命的な OutOfMemory エラーが発生し、それはすぐに滝の流れのようにクラスタの完全な崩壊につながります。

一般的なルールとして、クラスタ全体のメモリ使用の問題を検知することは防衛策としての最後の一線です。以下のセクション D では、個別の名前付きキャッシュの容量をもっと細かく制御する方法を検討します。

よくあるタイプの監視画面では、ヒートマップを利用してクラスタ内のメモリ使用状況を表示します。次の画面では、各ノードは矩形で表され、その矩形のサイズまたは領域は、そのノードに割り当てられたヒープ・メモリを表し、色はそのノードで現在使われているヒープ・メモリのパーセンテージを表しています。

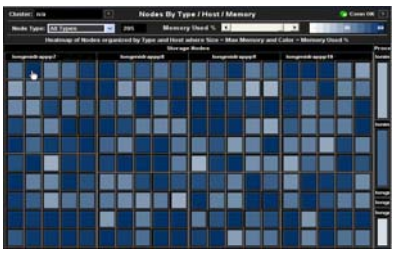

図 4 – 全ノードのメモリ使用状況を表示するヒートマップ

この図では、プロセス・ノードはストレージ・ノードとは別のグループで表示されていることに、注目してください。ノード停止に関しては、プロセス・ノードとストレージ・ノードであてはまるルールが異なることがよくあります。

このレベルでのノードのメモリ使用状況の監視は、見た目ほど簡単ではありません。JVM がレポートしたメモリが正確であるとは想定できないのです。このタスクは、Java の組み込みガベージ・コレクションの影響によってややこしくなります。

2 つの問題が事態を不鮮明にします。まず、JVM が "used" としてレポートするメモリには除去済みだがまだ削除されていない（ガベージと見なされる）オブジェクトが含まれており、実際にはどれだけメモリを使っているかを正確に言うのが難しくなっています。2 つ目に、JVM がガーベージ・コレクションの操作を行おうと決めたとき、そのノード内のすべてのプロセスは一時的に停止し、他のノードとの通信に長い遅延を生ずることがあり得ます。

次のトレンド・グラフは、サンプル・ノードがレポートしたメモリ使用状況を表示しています。のこぎりの歯型のトレースが、このノードでのガベージ・コレクションの影響を際立たせています。

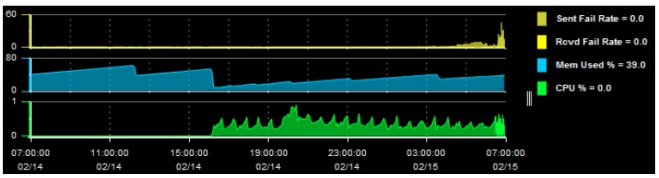

図 5 – シングル・ノードのメモリ使用状況（青）を表示するグラフ

一定の時間の間、レポートされたメモリ使用状況は、このノードへのオブジェクト追加あるいは削除にしたがって定期的に増えています。しかし、ガベージ・コレクション・イベントの度に、レポートされる使用量は以前と同じくらいのレベルに落ちます。これが示すことは、メモリ使用は本質的に水平であり、レポートされるほどには増減していないということです。

よって、図4. で表示されたメモリ使用状況のヒートマップは大変な誤解を招きます。そこでは、ノードは使用率に広い幅があるように見えます。レポートされたメモリは、のこぎりの歯のどの時点でクエリを行ったのかに、完全に依存します。

これではアラートを生成するレベルの設定が難しくなります。多くの場合、ガベージ・コレクションを行うまで、JVM は、割り当てられているうちの非常に高いパーセンテージまで、メモリを増えるがままにしていることがあります。問題があることを確実に知る唯一の方法は、のこぎりの歯の底のレベルをアラートのトリガとして使うことです。のこぎりの歯の底が特定のレベルに達した時にアラートを生成し、ユーザにクラスタ内のメモリ増加の問題を知らせることができます。

JVM の Platform Bean は、Java のガベージ・コレクションの振る舞いについて詳細な情報を提供します。もう一つの役に立つ情報は、ガベージ・コレクションの実行にかかる時間です。ガベージ・コレクションの時間が非常に長いのは別の種類の問題を示していることがあります。キャッシュに小さなオブジェクトがたくさん含まれており、ノードに大きなヒープ空間が割り当てられている場合、ガベージ・コレクションに非常に長い時間がかかり、通信に遅延が生ずる結果となります。これは、アラート生成に使えるもうひとつの指標です。

メモリ低下の検知とアラートは、防衛策の最後の一線として役に立ちます。しかし、それはそのメモリがどこで使われているかという手がかりは提供してくれません。幸い、Coherence ではクラスタ内で管理されている個別キャッシュのデータに対してより細かい制御が行えるようになっています。

D. *キャッシュ容量の監視と制御*

クラスタ内の各キャッシュを構成して、保持できるデータ量に制限を設けることができます。そのキャッシュにさらにオブジェクトを追加しようとすると、古いオブジェクトが退去させられる結果になります。

この機能には二重の目的があります。1つは、特定のオブジェクトのグループが利用できるメモリ量に制限を設けることです。すべてのキャッシュをこのように構成すると、事前にメモリの最大使用量を制御できます。もう一つは、これが頻繁にアクセスするデータにすばやくアクセスする機構も提供していることです。そして、クエリされるオブジェクトがキャッシュになければデータベースから取り出して、キャッシュ内の別のオブジェクトと置き換えます。

Cache MBean は、キャッシュ内の現行使用メモリとメモリ制限の両方を返します。この指標は「ユニット」として参照しますが、これは、オブジェクト数かバイト数のいずれかでレポートするよう構成できるからです。バイトに設定しておくと、それは消費メモリ量を表すため、最も有用です。

次の画面では、いくつものキャッシュでの使用メモリ量（ユニット）に加え、そのハイ・ユニットすなわち最大値を棒グラフで表示しています。キャッシュによっては容量に近づいていますが、他のものはそうではありません。

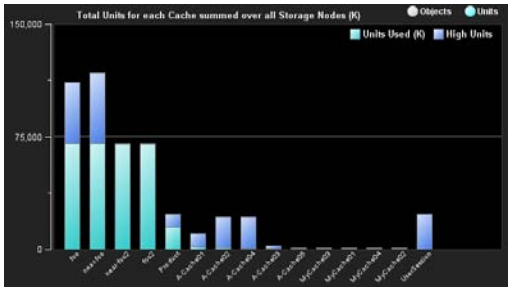

図 6 – キャッシュのユニットとハイユニットを表示する棒グラフ

ハイ・ユニットの設定を活用することにより、各ノードで利用可能な JVM ヒープ空間をセグメント化し、注意深く制御することができます。各キャッシュには特定のサイズ制限を持たせることができ、それを監視することができます。もちろんこの機能が適切となる利用事例がどんなアプリケーションにもあるわけではありません。ときには組み合わせるのが有用です。キャッシュによっては自由に増加できるようにしておくことができます。

典型的な使い方は、キャッシュをデータベースに対するバッファとして機能させることです。この場合、バッファとして機能させるために一定量のスペースを割り付けておくとよいです。一旦スペースがいっぱいになったら、オブジェクトは退去させられます。あまりたくさんの退去が起こり過ぎないように、監視してアラートを出すことが望ましいです。

次のヒートマップは、ハイ・ユニットがノードの利用可能メモリを如何に効果的に分割化しているかの見本です。

図 7 – ノード上のキャッシュのパーティショニングを表示するヒートマップ

このヒートマップでは、ラベルの付いた各ノードは1ダースほどのキャッシュのデータを保持しています。箱の大きさはそのキャッシュの総容量を表し、色はその容量の何パーセントが埋まっているかを示しています（濃紺は 100%を意味します）。

E. *クラスタ通信エラー*

Coherence クラスタのメンバは、Coherence TCMP という UDP ベースのプロトコルで通信します。TCMP はもともと Tangosol Communication Management Protocol（タンゴソル通信管理プロトコル）の略です。これは UDP のスピードに TCP の信頼性を組み合わせたものです。

これまで遭遇したことのあるクラスタの崩壊状況の症状のほとんどすべてはクラスタ・ノードの通信エラーです。このため、TCMP のパブリッシャ/レシーバのエラーは、最も重要な指標の1つとして追跡するべきものです。

JMX は、クラスタ・ノード毎にパブリッシャ/レシーバの成功率のデータを提供していますが、この指標はクラスタ・ノードが起動して以来の平均値として提供されるため、無視するべきです。今エラーが起こっているかどうかは、教えてくれません。より良い方法は、最後の収集インターバルでの送信と受信のパケット総数から差分を取り、次にパーセンテージを計算して、（RTView のような）監視システム内でこの指標を計算することです。崩壊状況でよく見られる高いエラー率は、パケット・エラーで 40% から 50% にまで上ります。

いったい何がこのエラーの原因でしょうか？興味深いことに、こうしたエラーがネットワークの問題から生ずることはまれです。マルチキャストを使っているので、たくさんの情報がノード間でブロードキャストされます。よくある筋書は、ストレージ・ノードがクラスタを離脱するときです。この場合、残りのノードはお互いに通信して、あるノードの応答がなくなったので「停止した」と宣告すべきであることを見つけます。一旦そうなると、他のノードにバックアップされていたデータは、プライマリ・データに変更しなければなりません。そして、バックアップ・データを再生成して他のクラスタ・ノード間に分散しなくてはなりません。データの再分散が何回か起こり得ます。通常これが、ネットワーク活動の短い嵐を呼び起こします。ノードはデータの処理に忙しいので、パケットのパブリッシャとレシーバはタイムアウトし、（パケット・エラーの結果）パケット再転送のレポートが発生します。次に元のノードがクラスタに再び加わると、さらにもう1回データ分散がトリガされます。Coherence 3.6 の Quorum ポリシーにより、ユーザはこの再分散を最小限にするようクラスタを構成できます。

通常、ノードがクラスタを離脱するとき、データの再分散が行われている間に長くても数秒間のパケット・エラーの短い炸裂があります。この状況は大抵一時的なものであり、ノードがメンテナンスのためにダウンした結果という、よくあることです。従って、このデータに対して定義したアラートはいずれもこの短い炸裂を考慮して、重大なアラートをトリガしないようにしなくてはなりません。同様に、クラスタ起動時とキャッシュのウォームアップ処理中に通信エラーが見られるのは普通のことです。

以上より重要な捉えるべき別のケースがあります。それはガベージ・コレクションに関するものです。

クラスタ内の活動パターンによっては、ガベージ・コレクションが問題になる場合があります。データがキャッシュに入れられ、すばやく削除される場合、ガベージが蓄積されていきます。エントリ・プロセッサやインデックスの構築と、更新やライト・ビハインド・キューなど、他のクラスタ操作もガベージを生成します。

JVM はこれに応じて定期的にガベージを収集し、これを削除しなくてはなりません。このプロセスは非常に時間を消費することがあります。たくさんの技術が導入され、これを最適化して、使用ケースバイケースに異なる振る舞いをさせるオプションが提供されてきました。

しかし、ガベージ・コレクションが発生しているとき、ノード内の他のすべての活動は停止します。つまり、パケット転送はタイムリーには処理されず、結果として大量に通信エラーが増えます。

このように、通信エラーの監視はクラスタ・エラーを防ぐのに欠かせないステップです。このパターンでは通常、まず通信エラーが起き、次にガベージ・コレクションが十分な仕事をせずにガベージが削除されていないため、ガベージ・コレクション後のメモリが増加します。メモリがなくなると、やがてノードの停止を招きます。その結果、データの再分散が行われて、さらにパケットのロスと遅延が発生し、クラスタ全体がダウンする結果になります。これもまた、Coherence 3.6 の Quorum ポリシーでは、この撹拌状態を軽減し、さらにクラスタがダウンへの道を辿り続けるのではなく、フリーズすることもできるように設計されています。

通信エラーを効果的に監視することはつまり、その時間を設定可能なようにしておく必要があり、そうすることで一過性の無害な再分散でアラートを発生させないようにすることです。一定レベルを超える長時間の通信エラーは、クラスタで問題が起ころうとしていることの確かな兆候です。

次の画面で示すのは、ガベージ・コレクションによってパケット・エラーがあまりに多数引き起こされているときに見られる、典型的な通信パターンです。この場合、アラート生成にトリガ・レベルを設定できるようにしておけば、それをあとで調査して修正できます。

図 8 – パケット・ロスの突出を起こすガベージ・コレクション（青）

通信エラーの監視で警告を出すことはできますが、通信エラーが増加するころには、すでにクラスタ性能に悪い影響を与えているかもしれません。このタイプの問題を検知するもうひとつの方法は、

ストレージ・ノードとプロセス・ノードをまたがったリクエストの分散を観察することによって、クラスタ内のデータ・アクセスを監視することです。

F. *ホット・キー、1ノードへの過剰なリクエスト*

よく見られる興味深いケースとして、「ホット・キー」の問題に絡んだものがあります。Coherence は、データ・リクエストがデータ・セット全体で（ほとんどの部分に対して）均等に分散されているようなデータ・アクセス・パターンの扱いには長けています。しかしときに、データの性質によっては特定の時間間隔内であるオブジェクトが他のオブジェクトよりも多くアクセスされる場合があります。

同じデータに何度も何度もアクセスすることは、そのオブジェクトのオフィシャル・コピーを持っている1ノードが繰り返しアクセスされることになります。そのノードは1度にサービスできるのは1つのリクエストだけです（スレッドを追加で割り当てれば増やせますが、それでも限界があります）。つまり、他のノードは自分のリクエストがサービスされるのを待たなくてはならないということです。そのノードがあまりに長い間「応答なし」と受け取られると、ノードはクラスタの他のメンバから「停止した」ものとして宣告される場合があります。するとそのノードは、クラスタから追い出されます。

一旦ノードが追い出されると、データの再分散が発生しますが、そのデータを含む新しいノードはリクエストへの応答を開始しなければならず、やがてそれも同様に追い出されます。

次のチャートでは、クラスタ内の全プロセス・ノード（上）と全ストレージ・ノード（下）の CPU レベルの履歴を表示しています。期間の半分を過ぎたくらいのところで、1つのノードが高いリクエスト数を、従って高い CPU レベルを、見せ始めます。これはその1つのノードに格納されている同じデータに対して、全プロセス・ノードがリクエストしているためです。

図 9 – プロセス・ノードとストレージ・ノードの CPU レベルの履歴

こうした筋書ではよく、CPU レベル以外にも他の数々の兆候が現れます。たとえば、ネットワーク通信エラー率は上がることがよくあります。このクラスタの負荷がもっと重い場合、そのノードはクラスタから追い出されることになります。

残念ながら Coherence は、特定のキーに対する要求数を判断する方法を提供していません。もちろん、アプリケーションに JMX を仕込んでこのタイプの情報を提供することはできます。アプリケーションの設計に監視を組み込むことは良い習慣ですが、残念ながら普通これは後になって思いつくことです。

V. まとめ

Coherence では、Coherence のクラスタリング・プロトコルの整合性を脅かしクラスタ自身の整合性を脅かすような常習的な状況を監視し理解することが重要です。インメモリ・システムの障害は全データを失う結果になるため、壊滅的なクラスタ・エラーになることがあります。あるユーザは、これは事故現場に破損した車のない自動車事故で、何が起きたかを理解しようとするようなものと表現されています。

TCMP パブリッシャ/レシーバ・エラー、ガベージ・コレクションの中断時間、ガベージ・コレクション後の利用可能メモリといった重要な運用指標を監視することにより、非常に切迫した障害を回避できます。また、監視指標を収集して永続化することで、何かがおかしくなった場合の事後分析にも利用可能になります。

参考文献

[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

本書は、英語原文の Technical Paper “Detecting and Alerting on Fault Conditions in an Oracle Coherence Distributed Caching System” を和訳したものです。

株式会社 SL ジャパン

〒107-0062 東京都港区南青山3-8-5

アーバンプレム南青山 3階

Tel. 03-3423-6051 info@sl-j.co.jp

www.sl-j.co.jp