J-PARC 運転データアーカイバにおける HBase/Hadoop のバージョンアップ対応及び ZooKeeper を使ったデータ収集ツールの冗長化

UPDATING HBASE/HADOOP IN THE J-PARC OPERATION DATA ARCHIVER WITH MAKING THE DATA COLLECTION TOOL REDUNDANT BY ZOOKEEPER

池田浩[#, A, B], 菊澤信宏[A], 吉位明伸[C], 加藤裕子[A]
Hiroshi Ikeda [#, A, B], Nobuhiro Kikuzawa[A], Akinobu Yoshii[C], Yuko Kato[A]
[A] Japan Atomic Energy Agency
[B] Visible Information Center, Inc.
[C] NS Solutions Corporation

Abstract

The Linac and the RCS in J-PARC provide enormous operation data to control their equipment, and we have been accumulating the data into a PostgreSQL database system, while we are planning to replace the storage with HBase/Hadoop. HBase is so-call NoSQL, specialized for big-data with scalability to the data size. HBase is constructed on a distributed file system provided by Hadoop, and both HBase and Hadoop manage their redundancy property by ZooKeeper, a service to coordinate nodes in distributed environment. Prior to starting our system toward full-scale operation, we have re-updated HBase/Hadoop in our cluster, for reasons including that the old version of HBase has been already out of support; HBase API has started to specify backward compatibility issues so that we would be able to easily fix our code to the new versions; It will become much more difficult to update our cluster once we start our system and are burdened with stored data which should be never lost. Following the update, we have fixed our kick-start scripts, and monitoring scripts to check hardware and software's health. We also have updated our management tools that we created. On the other hand, it is true that HBase/Hadoop can strongly hold their data, but that makes sense just when the data is coming without lost. Accordingly, besides the update issue, we have made our data collection tool redundant through ZooKeeper, taking after HBase/Hadoop, and the tool now automatically does actions against system troubles even when you just deploy the tool in multiple nodes. In this presentation, we are reporting the contents around updating HBase/Hadoop, and the details of making the data collection tool redundant, with reporting the revealed issues in the way.

1. はじめに

J-PARC の Linac 及び RCS では機器から得られるデータを PostgreSQL に蓄積しているが[1]、これを HBase[2]/Hadoop[3]を用いたシステムに変更するための準備を進めている。前回の発表では、クラスタを再設計・再構築したことを述べたが[4]、本格的な運用に先立ってバージョンアップを行った。また、HBase/Hadoop は同クラスタに配備される ZooKeeper[5]を用いて冗長化を実現しており、我々のデータ収集ツールもこれを用いて冗長化を行った。今回の発表では、これらについて述べる。

2. HBase/Hadoop のアップデートと設定

HBase/Hadoop 等を Table 1 に示すようにアップデートを行った。Java のバージョンについては、以前の HBase 0.96.1.1 では Java7 はサポート外だったが、HBase 1.0 系から Java 7 がサポートされ、逆に Java 6 はサポート外になっている。

Table 1: Software and Updated Versions

	Previous	Current
Java	6u45	7u80
HBase	0.96.1.1	1.1.2
Hadoop	2.2.0	2.5.2
ZooKeeper	3.4.5	3.4.6

2.1 キックスタート時の修正パッチ

HBase 等のアップデートにより一部の不具合が解消されたため、キックスタート時にこれを修正していたパッチを変更した。特に、PID ファイルが残ってしまうために、次回起動時に関係のないプロセスを殺す可能性があるという問題については、HBase には修正が入っている。一方、Hadoop や ZooKeeper に対しては修正が入っていない。

2.2 シェルスクリプトの環境変数

HBase, Hadoop ともにサービスの起動や操作、メンテナンスで使うシェルスクリプトにて用いられる環境変数が変更されており互換性がないため、改めて確認と設定が

ikedahi@post.j-parc.jp

必要になった。

その他、互換性のない大きな変更としては、コンポーネントが使用するポートの既定が、OS が自動的に割り当てるのに使うエフェメラルポートの範囲を避けるように変更されていることが挙げられる。ただし、Hadoop の一部は未だこの範囲のポートを使っている。

2.3 設定ファイル

既定の設定ファイルが用意されているが、完全に網羅しているわけではない。変更・廃止されたパラメータを知るには、コードを確認するか、実際に動かしてログを確認するくらいだろう。Web から参照できる HBase の JIRA（プロジェクト管理ツール）を見れば、恒常的に新しくパラメータが追加されたり廃止されていたりするのが見て取れる。

これとは別に、以前から存在するパラメータの幾つかを修正し、我々の用途により適切な設定を行った。幾つかを列挙すると、HDFS のデータの永続性に関してデータノードが受け取ったブロックの書き込みを閉じるときに sync する（dfs.datanode.synconclose）、データノードのエラーを起こす記憶媒体の許容数として 1 を指定する（dfs.datanode.failed.volumes.tolerated）、長期間のサービスの停止は許容できないため、ZooKeeper のタイムアウトとして 30 秒を指定する（zookeeper.session.timeout）ことが挙げられる。

2.4 監視用スクリプト

クラスタやソフトウェアのヘルス監視は Nagios[6]を利用している。一般的な項目に対しては正常・異常の判定をするスクリプトは用意されているが、そうでないものは自分で記述する必要がある。

HBase や Hadoop では、ステータス状態を標準出力に出力するための方法が用意されている。JMX 経由でさまざまなメトリクスを取得する方法なども用意されているが、分かりやすく手動での再確認が簡単である方法であることから、我々は前者の標準出力の結果を利用して監視用のスクリプトを作成している。HBase 及び Hadoop のアップデートにより、出力書式・内容が変更されており、監視スクリプトの修正が必要になった。

また、HBase 等の各コンポーネントが稼働している確認の一つとして、JDK の JPS コマンドを利用して該当するプロセスが OS に存在しているかを確認しているが、Java のアップデートにより root 権限で他ユーザのプロセスを検索できなくなった不具合のため、スクリプトの修正や実行許可の修正が必要になった。

3. ツールのアップデート

HBase の API は後方互換性を明記するようになったが、Hadoop や ZooKeeper と同時に使うこともあって、依存する 3rd パーティライブラリが 100 を超えるようになった。また、これらのライブラリがどの機能を使う際に必要になるかが明記されておらず、特に、クライアントの用途で使う場合に必要なライブラリがどれなのか不明瞭である。我々のツールにて、膨大な全てのモジュール及びライブラリに依存させることは、自由に使えるライブラリが限定されることもあり、また、異なるバージョンのライブラリも含まれており不具合の元であることからライブラリの依存関係や役割を調査し、クライアントの用途として必要なモジュール及びライブラリを選定し直し、30 個程度に絞った。

また、HBase が明記する後方互換性は、実装を見ると、例えば古い API はバッファリングが一律無効化されているなど実質的に使い物にならず、当てにならないこと分かった。このため、新しい API にてツールを書き換える必要があった。

3.1 PostgreSQL からのデータ移行ツール

PostgreSQL からのデータ移行ツールについては、HBase の通常の API を使ってデータを転送する他、Hadoop の MapReduce を用いたバルクロードを使ったツールも用意していた。しかし、単純にバルクロードを繰り返すと HBase の内部構造が歪になる問題がある一方、新しいバージョンの HBase では API 経由にてリージョン数が少ない場合はスプリットが積極的に行われる戦略が採られておりリージョンサーバの負荷分散が期待できること、また、テーブルの圧縮形式を適切に設定することで通常の API を使ったデータ挿入速度は格段に向上することが分かったことから、通常の API を使うツールを、アップデートされた HBase 用のツールとして再構築した。

3.2 運転データ収集ツール

運転データ収集ツールは、チャネルアクセス[7]を用いて加速器機器から運転データを取得し、我々のクラスタにデータを格納する。このツールに関しては、HBase のアップデートに対応する他、ZooKeeper を用いた冗長化を行った。これについては、後述する。

3.3 データ閲覧ツール

データ閲覧ツールについては、CS-Studio[8]（以下 CSS）のプラグインとして構築していたが、これを機に対応する CSS のバージョンを上げた。HBase 1.1.2 でサポートされるバージョンは Java7 であるため、これに対応するメジャーバージョン 3 で KEK 用にカスタマイズされた CSS のうち最新の CSS 3.2.16 を採用した。CSS 3.1 系とは基盤とするデータ構造が別のパッケージのものに取り換えられたことが判明し、これに対応した。

4. 運転データ収集ツールの冗長化

ZooKeeper を使って運転データ収集ツールを冗長化した。すなわち、ZooKeeper にてリーダに選出されたノードのみが、チャネルアクセスのモニタを登録し、これより収集したデータを HBase に転送する。

チャネルアクセスへの接続にはサーバ検索に時間を要するため、異常時に備え、リーダかそうでないかに関わらず、HBase やチャネルアクセスへの接続は完了しておく。モニタ経由でデータを取得するため、これによる無駄なデータ転送は起こらない。

厳密に言えば、冗長化を ZooKeeper だけに頼ると、ZooKeeper への一時的な接続障害によってリーダ権限を失ったことに気がつかない可能性から、いわゆる「スプリットブレイン」が発生する危険性がある。ここでは、我々

の用途では ZooKeeper を稼働しているノードでアプリケーションをこれと共に稼働させることを想定し、ZooKeeper への接続障害だけが発生する可能性が小さいこと、また、スプリットブレインが発生しても致命的な問題にはならないことから、複雑にすることを避け ZooKeeper に依存したコーディングを行った。

冗長化は、監視が伴わなければ、いずれサービスは停止する。ツールの稼働の監視は、ZooKeeper 上のエフェメラルノードの数を数えることとした。これを Nagios 経由で定期監視するスクリプトを用意した。

以下に、運転データ収集ツールの冗長化の作業にて判明した問題点等を記述する。

5. ZooKeeper に関する知見

5.1 ZooKeeper で利用可能な機能

ZooKeeper は、分散環境におけるノード間の調停を行うための手段を提供するサービスとして位置づけられるが、リーダ選出や排他処理そのものは ZooKeeper に元から備わっている機能ではなく、適時、ZooKeeper の機能を利用して実装することになる。

ZooKeeper が提供するものはおおよそ、znode と呼ばれるノードで構築されたツリー構造による簡易的なファイルシステムとなる。ノードには、リモートプロセスからのハートビートを受信している間だけ存在するエフェメラルノードや、指定したパスのプレフィックスに自動的に番号を付加して一意のノードを作るシーケンスノードがあり、同時に両者を指定できる。ZooKeeper は必ずしも全てのクライアントに対して最新のツリー構造を示せるわけではなく、ZooKeeper のドキュメントでは ZooKeeper が保証するものとして Consistency Guarantees として 5 つの条項が挙げられている。なお、ノード内に記述されたデータが最新であることはこの保証には含まれず、この用途に sync メソッドが用意されている。また、クライアントはノードに対しその変更に応じて 1 回限りの通知（watch イベント）を要求することができ、これらを基にして、リーダ選出や排他処理を実装していくことになる。

5.2 レシピ

ZooKeeper のドキュメントにはリーダ選出を含めた幾つかの設計・指針に関する情報が記述されており、「レシピ」と称されている。

ZooKeeper のドキュメントに記述されるリーダ選出のレシピは、サービスを明示的に正常停止することを想定しておらず、サービスの停止時に無駄なリーダ選出が複数回行われる可能性がある。これは、特に、リーダとして選ばれた際の初期化処理の負荷が高い場合や、リーダが再選出された際のデータの欠如や重複が発生する可能性がある場合は望ましいものではないだろう。結局のところ、リーダ選出や排他処理には汎用的な実装は用意できず、レシピで示される指針をそのまま使うのではなく必要に応じてカスタマイズして適用していくということだと思われる。

我々は、具体的には以下のようにレシピをカスタマイズした：自分がリーダに選出されたかどうかを確認するには、ZooKeeper のファイルシステム上の指定したディレクトリに含まれるノードのうち、辞書式に最も小さい名称のノードが、自分が登録したエフェメラルノードであるかどうかで判断する。このため、同ディレクトリに他よりも小さい通常の（すなわちエフェメラルでなくシーケンシャルでもない）ノードを作成することで、リーダ選出の機能を実質的に無効化する。

一見簡単そうに見えるが、前述したとおり、ZooKeeper は必ずしも全てのクライアントに対して最新のツリー構造を示せるわけではなく、作成した無効化用ノードが必要なときに可視になっていることをロジック上で保証する必要がある;リーダに選出されたかどうかを確認する際は同ディレクトリに含められる全てのノードを取得するため、この確認をトリガーする watch イベントの前に無効化用ノードを作成すれば、ZooKeeper の Consistency Guarantees よりこの無効化用ノードは可視になる。

このロジックは、既に選出された（あるいは無効化が watch イベントに間に合わず新たに選出された）リーダを未選出の状態に戻すものではないが、サービスを正常停止するのに利用するには十分だろう。さらに、この無効化用ノードを削除すれば本来リーダとなるべきノードに対する watch イベントが（もしあれば）トリガーされ、リーダ選出が再開すなわち実質的に有効化する。

無効化用ノードの名称としては、ピリオドで始まる文字列は辞書式におおよそ他の文字列よりも小さいため、例えば、“.disabled”などという文字列を選べばメンテナンス時に分かりやすいだろう。

5.3 コンストラクタでのスレッド開始

ZooKeeper は、コーディングレベルで幾つかの問題を抱えている。

ZooKeeper では Watcher と呼ばれるコールバックを使って、問い合わせと監視の設定を同時に行う。クライアント側のエントリークラス ZooKeeper では、コンストラクタに Watcher を渡せるが、ZooKeeper クラスがコンストラクタにてスレッドを開始するという典型的なアンチパターンを使っており、コールバックが ZooKeeper のインスタンスへの参照を得る前に呼び出される可能性を考慮する必要がある。コールバックの処理に ZooKeeper のインスタンスを必要とするならば、コールバックを一時的にブロックする方法が必要になる。HBase, Hadoop ともこれに起因する問題を回避するために試行錯誤を行って多くのコードを記述している。

このコールバックの一時的ブロックには、技術的には Java 言語仕様における final フィールドの semantics を利用すると、インスタンス生成後の同期のオーバヘッドを避けることができる。これは、インスタンスレベルのシングルトンを作成する際に使われるのと同じ Java 特有のイディオムである。

5.4 2 つの内部スレッド

ZooKeeper のクライアント側では内部で主に 2 つのスレッドを使用している。

一つは、サーバとの送受信を扱うスレッドであり、結果が得られるまでブロックされるユーザ用のメソッドをリリースするのも、このスレッドで行っている。コード内では SendThread というクラス名が付けられており、ミスリードに注意する必要がある。

もう一つは、ユーザから渡されたコールバックメソッドを呼び出すスレッドである。コード内では EventThread というクラス名が付けられている。ユーザスレッドを用いて問い合わせをした場合（つまり、コールバックメソッド中からの問い合わせでない場合）、問い合わせの応えに対する処理を行っている間に、コールバックが EventThread にて非同期に呼ばれる可能性があることに注意する。

さらに、コールバックは、サービスの終了間際には SendThread やユーザスレッドそのものを使って直接呼び出されることもあり、ロック内で呼び出していることからデッドロックに気を付ける必要がある。

2 つのスレッドを使っているのは、おそらく、コールバックメソッドからブロックするメソッドを呼び出しても、デッドロックにならないようにするためだろう。しかし、コールバックを正しく記述するのは、スレッドに関する洞察が必要になり難易度は非常に高くなっている。

5.5 特殊な接続状態

ZooKeeper のサーバへの接続が一定以上途絶えると、Expired 状態と見なされる。これはサーバ側の判断であり、次のクライアントの自動再接続時に通知される。これを通知された ZooKeeper クライアントは再接続したばかりのコネクションを切断し、終了状態に移行し、このインスタンスはもはや使い物にならなくなる。ZooKeeper のサーバに再び繋ぎたい場合は、新しいインスタンスを作成するところから始めなければならない。

ZooKeeper クラスのインスタンスが自発的に利用不可になるのは Expired 状態の他に、サーバへの接続権限不足からの AuthFailed 状態がある。この 2 つに限っては、他の状態通知と異なり、明示的に特別に扱う必要があるだろう。

6. チャネルアクセスライブラリの改良

チャネルアクセスは、サーバに対して該当するチャネルが存在するかどうかの問い合わせに UDP を使う。一般的に、ブロードキャストにて同一ネットワーク上に存在するサーバ群に対して、チャネルの所在の問い合わせを行う。

我々は運転データ収集ツールを冗長化したが、これが意味を成すためには管理可能な複数台の機器にこれを配備する必要がある。機器自体のヘルス管理を行っている我々のクラスタのノード上に配備するのが最も相応しいが、接続予定のチャネルは複数のネットワークに跨っている。CA ゲートウェイはこの目的のためには運用・保守されておらず、ディレクティッドブロードキャストも許可されていない（ブロードキャストの許可変更は J-PARC 全体のネットワークのセキュリティの根本に影響を及ぼすだろう）。

JCA[9]など一般的に使われているチャネルアクセスライブラリでは、ローカルブロードキャスト以外にもチャネル検索先アドレスを指定することができる。しかし、指定されたアドレス毎に対して全てのチャネルに関する問い合わせを行うため、膨大なチャネル数を扱う J-PARC では、チャネルが存在する可能性のあるサーバを単に全て列挙して指定するのは、実用的な時間で接続処理を完了することは期待できない。

チャネルアクセスのプロトコル上では、アドレス毎に全てのチャネルを問い合わせることは必須ではなく、チャネルごとにそれが存在するはずのサーバに UDP パケットを投げれば十分である。このため、チャネル名のパターンでチャネルの検索先サーバを指定できるように、我々が使用するチャネルアクセスライブラリを拡張した。

ただし、J-PARC ではチャネルとサーバの関係は統括管理されておらず、構成変更や障害対応等で自由に変更されることもあり、上記の拡張だけでは対応が不十分であった。このため、将来的に使用される可能性のあるサーバを含めた広い範囲のサーバを検索対象として指定できるように、さらなる改良を行っているところである。もともとチャネルアクセスでは大量のブロードキャストを主体としたチャネル検索を行っており、これを許容するネットワーク上でこの方法によるネットワークの負荷が問題になることはないだろう。

7. 今後の対応

現在、上記で挙げたツールを使って、PostgreSQL からのデータ移行を始めたところである。およそ Linac のデータ 1 日分を 5 分で移行できており、単純計算では 1 日半で 1 年分を移行可能となる。

運転データ収集ツールについては、上記に述べたようにさらに修正を要する。また、現在、PostgreSQL にデータを収集しているチャネルの対象は、別途データベースに格納されており、ここから設定ファイルを自動生成するアプリケーションを作成する予定である。

運転データの格納後は、Hadoop の MapReduce を用いたデータマイニングにより、複数のデータの関連性から J-PARC の障害発生の予測検知ができないかを検討したい。

参考文献

[1] S. Fukuta *et al.*, “Development Status of Database for J-PARC RCS Control System (1)”, Proceedings of the 4th Annual Meeting of Particle Accelerator Society of Japan, August 2007.
[2] http://hbase.apache.org/
[3] http://hadoop.apache.org/
[4] H. Ikeda *et al.*, “Improvement of the J-PARC Operation Data Archiver Using HBase/Hadoop”, Proceedings of the 12th Annual Meeting of Particle Accelerator Society of Japan, August 2015.
[5] http://zookeeper.apache.org/
[6] https://www.nagios.org/
[7] http://www.aps.anl.gov/epics/docs/ca.php
[8] http://controlsystemstudio.org/
[9] http://epics-jca.sourceforge.net/jca/