MADOCA II データー収集と蓄積システム
MADOCA II DATA ACQUISITION AND STORAGE SYSTEM

山下明広、籠正裕、酒井久伸
Akihiro Yamashita*, Masahiro Kano, Hisanobu Sakai
Japan Synchrotron Research Institute/SPring-8

Abstract

In SPring-8, we are constructing MADOCA II, next generation accelerator control framework. It will be installed in the spring of 2014. We describe the part of the data acquisition and the storage system of MADOCA II. MADOCA was built on the bases of ONC-RPC for communication between processes and a relational database for data management. We designed the new framework with the long experience on MADOCA. We employ Zeromq messages packed by Messagepack for communication. NoSQL databases, Redis and Apache Cassandra, store log data. We obtained a high performance, highly reliable, well scalable and flexible data management system. In this paper, we will discuss requirements, design, implementation and the result of the long run test.

1. はじめに

SPring-8 では 1997 年のコミッショニング以来、加速器とビームライン制御フレームワークとして MADOCA (Message And Database Oriented Control Architecture)[1] を使用している。

MADOCA はその名前が示すとうり、データーの管理をデーターベースに依存している [2]。単一の Relational Database Management System (RDBMS) が機器の管理、運転パラメーターの管理、およびログデーターの保存に使用されてきた。ログデーターベースは加速器とビームラインの全ての機器のログデーターを常時記録することにより安定運転と高度化に寄与してきた。

そのなかで、RDBMS をログデーターの管理に使用することに以下のような問題点が見えてきた。

1 つは新しいデーターを登録する際に手間がかかることである。MADOCA では、1 つの RDBMS のテーブルの 1 つのローに多数のデーターを入れる形式をとっているため、新しい信号を登録する際にはテーブルの新規作成をおこなわなければならない。1 つのローに多数の信号を入れるのはデーター挿入時の SQL 命令を減らし、書き込み性能を向上させるためである。

また1 つのローに多数のデーターを入れるという性格上、それらを同一時刻に取得せざるを得ず、データー取得の時間的柔軟性に欠けると言う問題点もあった。

性能上の問題も見えてきた。RDBMS はデーターの一貫性を再重要視するため処理は同一のメモリー空間内で行うという原則がある。このため性能向上はメモリーを共有したマルチプロセッサーシステムの性能向上に頼らざるを得なくなり、性能向上のためには、マルチプロセッサーのコア数を増やすか、プロセッサー自身の性能向上が必要である。さらに信頼性向上のためには、ディスクを共有化したクラスター構成か、ハードウエア的にコンポーネントを二重化した構成を取る等の方策がとられる。SPring-8 では以前は前者の方法を採用していたが、クラスター構成特有の管理の複雑さやクラスター切り替えの際の空白時間の問題があり、後者に切り替えた。それによりクラスター構成の問題点は克服されたもののハードウエアが高価になり性能向上には多大なコストがかかる問題点がでてきた。

また現 MADOCA のデーター収集系は ONC-RPC によりサーバーとクライアントが密結合されていたため、相互の依存性が高くまた OS(Unix 系に限定) や言語環境 (C に限定) に制限があった。

これらの問題点を解消し次世代の SPring-8 の運転に対応するため、MADOCA II の開発の一環としてデーター収集データー蓄積のシステムも一新すべく開発をおこなった。新システムの設計方針としては以下のとうりである。

- データー管理が容易であること、信号の登録に際してほとんど手間がなく行えること。
- データー蓄積単位が 1 信号で独立していること。柔軟性と管理性を向上させる。
- 時系列データーの蓄積、取得に適していること。
- スケーラブルであること、データー容量、性能向上を低コストで実現出来るものであること。
- 現在の MADOCA システムからの移行が容易であること、MADOCA は信号にヒューマンリーダブルな名前を命名することにより制御を行っているが、新システムでもそれを採用すること。
- 高信頼であること、データーベース、データーパスとも複数化し 1 つの機器の故障が致命的システムダウンに至らないこと。
- データー収集系が疎結合であり、相互依存性を極力排除する. オペレーティングシステムや言語環境にも依存も極力少なくする。
- データーベース構成はパラメーターは現在の RDBMS を使用し、ログデーターベースを改変する。

2. 設計

2.1 データー収集系

データー収集系の概略図を Fig. 1 に示す。SPring-8 の機器制御系はいわゆる 3 層標準モデルに従っていて、制御対象機器側に置かれた組み込みコンピュータとネット接続されたサーバーコンピュータによって構成される。組み込みコンピュータから Zeromq [3] メッセージの形式

* aki@spring8.or.jp

によりデーターが複数の中継サーバーに組み込みコンピュータのタイミングでプッシュされる。中継サーバーはメッセージを複数の書き込み用プロセスに転送する。

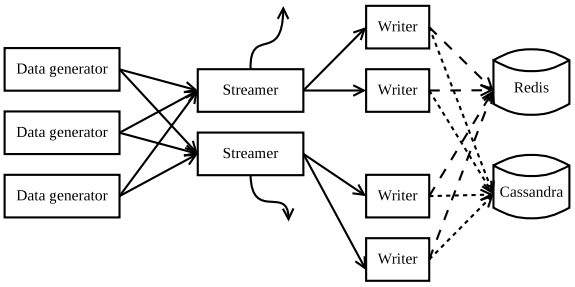

Figure 1: データー収集系　実線は Zeromq　Push/Pull、曲線は Zeromq Pub/Sub、破線は Redis API、点線は Thrift RPC。

Zeromq は様々なメッセージングパターンを選択できる多言語多 OS 対応の非同期メッセージングライブラリーである。ここでは組み込みコンピュータから中継サーバーと、中継サーバーと書き込みプロセス間に Push/Pull パターンを使用している。中継サーバーはメッセージを Publish(出版)/Subscribe(購読) パターンでメッセージを送る機能も備えており、購読予約をしたプロセスはデーターをリアルタイムで読み込める。Push/Pull パターンは 1 個の送り手から複数の受信者にメッセージをラウンドロビンで送るパターンで、ロードバランシングが容易に実現できる機能を持つ。さらに送り手は受信者のトラブルを検知することができ、トラブルがあった受信者を避けてメッセージを送ることができる。 Zeromq のこのパターンを使用し、複数の中継サーバーを使用すれば耐障害性を持ったロードバランシングが実現できる。

中継サーバーは複数の書き込みプロセスにメッセージを送る。後述のデーターベースへの書き込み API は、同期型なので、書き込みの間メッセージの受信ができなくなり、メッセージの待ち行列に滞留が起きる可能性がある。これを防止するため、複数の書き込みプロセスに作業を分散させることによりこれを防止している。

また、データー収集プロセスから直接書き込みサーバーに接続せず中継サーバーを使用するのは、Zeromq のソケット管理上の要請と組み込みコンピュータからの接続数を減らし管理を容易にするためである。

Zeromq の Pull ソケットに対してはいつでも複数のクライアントから接続が可能である。したがって任意の時間でデーター収集プロセスを起動し、データー収集を開始または停止することができる。

2.2　データーベース系

データーベースは永久保存用データーベースと現在値専用のキャッシュサーバーで構成される。2 つに分離した理由は永久保存用データーベースと現在値用のデーターベースは要求される機能が異り、1 つのデーターベースシステムではそれを実現できないためである。

永久保存用データーベースに要求されることは、まず書き込み性能である。現 MADOCA とは異なり、信号 1 つにつき 1 回の書き込み命令を実行するため、現在の毎秒 7800 信号の書き込みを十分越える性能を要求した。時系列でデーターを扱いやすい形式に対応していること、また挿入されるデーターが非定型であることスケールアウトにより性能向上が可能なこと、分散された複数のサーバーによって構成されること。データーの多重化ができることまた耐障害性のため、各ノードが平等でマスターを作らずの構成されることである。

Google の BigTable[4] 発表以来大規模な NoSQL (Not only SQL) データーベースが web サービスの分野で開発が盛んにおこなわれるようになった。 これらは高性能で、安価なサーバーにデーターを分散しスケールアウトが可能でデーターにも冗長性が持たせられる実装が多い。

われわれが選択したのは Apache Cassandra [5] でカラム指向データーベースとよばれる NoSQL データーベースである。 これはわれわれが調査した限りでは上記の条件を満す代表的なデーターベースである。 これをテストの上実用に十分耐えられるものと判断した。

Apache Cassandra はデーターを複数のノードに分散させる。そのため CAP 定理 [6] のうち可用性と耐分断性は満たすものの、一貫性は保証されない。 Cassandra では最終的には一致するもののデーター挿入の初期段階においてはノード間でデーターが一致しないということがある。Cassandra ではこれを結果整合性と呼ぶ。われわれの測定では 6 ノードでデーターを 3 重化した状態でデーターを取得した場合、矛盾が起きるのは挿入後 1 秒程度以内である。

この対策としてキャッシュサーバーを用意し、Cassandra とキャッシュサーバーにデーターを並行して書き込み最新値はキャッシュサーバーから得るという方式で Cassandra の結果整合性を補償する。

キャッシュサーバーとして Redis [7] を採用した。キャッシュサーバに一要求されるのは書き込み、読み込みとも高速で行えるものであることである。書き込みは Cassandra 並みであることが必要であるが、SPring-8 制御では最新データーを頻繁に要求する大量のクライアントからのクエリーを処理する必要がある。また 1 つのクエリーで多数の信号の値を取得できる機能も必要である。われわれはインメモリー型のキーバリューデーターベースである Redis はこの要求を満たすことがテストの結果わかった。キャッシュサーバーとして以前から人気のある memcached[8] と比較しても値に 1MB の壁が無い、ストアドプロシージャのような組み込みのアプリケーションが使用できるなどの利点がありこれをテストした結果採用することになった。また mget 命令により一度に大量のデーターを高速に取得できる。

3．実装

3.1　メッセージ構造

組み込みプロセスから中継サーバーに送られるメッセージは 3 つの Zeromq のマルチパートメッセージで構成される。第 1 のメッセージはキーバリューメッセージの 2 文字の接頭辞+信号名+終端文字 (:) 第 2 のメッセージはメタ情報を Messagepack [9] でパックされたマップデーター。第 3 のメッセージも Messagepack でパックされたデーターである。図 2 にメッセージの例を示す。第 1 のキーにより中継サーバーから信号名で subscribe

が可能になる。また最初の 2 文字のキーワードはそのメッセージのタイプを示すためで、ログデーターの場合は LG としているが、他の用途には別のキーワードを使用する。例えばアラーム発生のメッセージなどが収集したい場合には AL というような接頭辞をつければ同じ信号名でも使い分けが可能である。

メタデーターとデーターを別のメッセージにしたのは、メタデーターのみで判断分岐が可能なようするためである。現在のところはデーター取得時刻のみがメタデーターとして登録されているが、マップ形式なので将来の項目追加も容易である。

Messagepack は自己記述型のバイナリ形式のオブジェクトシリアライザである。これは JSON 形式をバイナリ化と言ってよく、JSON で表現できるオブジェクトはこれで高速かつでシリアライズ/デシリアライズすることが可能である。また出力の文字列も JSON でエンコードされたものより短い。

3.2 コンフィギュレーション設定ファイル

このシステムでは多くのホストコンピュータ中で多数の関係するプロセスが複数の通信ポートを使用する。それらを統一的に管理するため、JSON 形式による設定ファイルで全系での統一をおこなっている。

3.3 中継サーバー

中継サーバーは Python でプロトタイプを制作し、動作性能を確認後、C++で実装した。機能はごく単純で Zeromq の Pull ソケットから受け取ったメッセージを Push ソケットと Publish ソケットに送っているだけである。トラブル時の再接続などの機能は Zeromq ライブラリーに任せてある。1 つの中継サーバーは複数の書き込みプロセスに Push/Pull パターンを使用して接続されている。

3.4 書き込みプロセス

書き込みプロセスもは Python でプロトタイプを制作し、動作性能を確認後、C++で実装した。機能として受信したメッセージを Redis と Cassandra に並行して書き込むこのため内部で構成は受信スレッドと 2 つの書き込みスレッドで構成される。受信スレッドは Zeromq の pull ソケットから受け取ったメッセージを Zeromq のプロセス内通信で publish する (Fig. 2)。2 つの書き込みプロセスはそのメッセージを subscribe し、メッセージを Redis、Cassandra 用に加工し、書き込む。Cassandra は Thrift RPC を使用する。受信スレッドと書き込みスレッドの独立性が高いため、例えば Redis,Cassandra 以外のデーターベースに書き込む場合でも、専用のスレッドはメッセージを subscribe するだけなので、受信スレッドには何の変更が必要無い.

3.5 キャッシュサーバー

Redis はその特長としてリスト、ハッシュや集合など豊富な型システムが選べるが本実装では単純な文字列 1 つを値として挿入する。

複数のデーターストリームを持つ本データー収集系では必ずしも組み込みのサーバーから送られたメッセージが時間順序通りキャッシュサーバーに到着することは保証できない。対策をしなければ、最新値の後に古い値

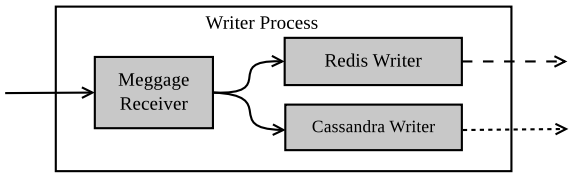

Figure 2: 書き込みプロセスの内部構造。灰色箱はスレッド。メッセージを受け取ったスレッドは Publish/Subscribe パターンを使用してデーターベース書き込みスレッドにメッセージを中継する。

が上書きされる恐れがある。この対策のために Redis の機能である組み込みスクリプティング機能を使用した。書き込みスレッドは書き込みの際に Redis の set 命令ではなく Lua 言語で書かれたスクリプトを実行する。このスクリプトは Messagepack でエンコードされた文字列をデコードして、時刻を現在ストアしているものと比較し、それが新しければ上書きする。

キーはメッセージのキー部分と同じ。バリューは時刻と信号値をまとめて Messagepack エンコードされた文字列である。

1 台の Redis の性能が飽和した場合にそなえて、複数の Redis サーバーで並列処理をおこなった。各キーのハッシュ値に対応したサーバーにデーターを振り分ける。

3.6 永久保存サーバー

カラム指向型データーベースである Cassandra で効率良く時系列データーを扱うため 1 つの信号について 1 日分のデーターには 1 つの行キーを与える (Fig. 3)。1 つのイベントデーターの追加には、時刻をカラム名としたカラムを追加する。という形でデーターを蓄積する[10]。この方式は読み込み、書き込み性能は良いものの、1 つのイベントデーターが 1 つの文字列に制限される。そこで、複雑なデーター構造は Messagepack 化した文字列で挿入する。この方法では信号名+ 日データーを新たに定義することなく書き込め、信号登録にかかる手間を大幅に低減することができる。

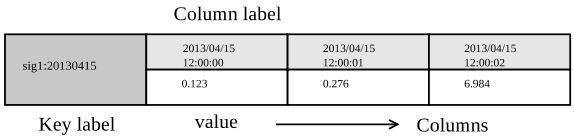

Figure 3: Cassandra のデーター構造。ある信号に対する時間データーはコラムを増やして収納する。信号はキーを増やすことで登録する。

4. テスト

4.1 書き込みテスト

書き込みの安定性をテストするため,Table 1 のような構成で書き込みの長期間テストを行った。このデーターは実際に SACLA で使用されている全信号とそれらのデーター型を模擬したものである。ただしデーター書き込み周期は現行より高速である。この構成で 3 ヶ月間の長期書き込み試験を行った。

Table 1: テスト パラメーター

CPU	Intel Xeon X3470
	2.93GHz 4Core
OS	CentOS 6.2 64bit
Redis version	2.6.10
Cassandra version	1.15
Java	OracleJavaVM 1.6.0
データー生成プロセス数	240
書き込み信号数	47397
書き込み速度	1Hz
平均キー長	38.7 bytes
メタデーター長	13 bytes
平均データー長	9 bytes
中継サーバー数	3
書き込みプロセス数/中継サーバー	8
Redis プロセス数	4
Cassandra サーバー数	6
Cassandra データー冗長度	3

データーを 3 重化し信頼性を向上させた。この機能を確認するため Cassandra サーバーを 1 台シャットダウンさせるなどの試験をおこなったがシャットダウン時やその後のリカバリー時でも書き込みに影響は出なかった。

非同期型書き込みという性格上書き込み遅延時間測定はおこなわず、データーの欠損を見たが、この間のデーターの欠損はなかった。

4.2 読み込みテスト

読み込みテストは、実運用時の状態に近づけるため書き込みテストと並行して行った。Table 2 に Redis の読み込み速度の測定結果を示す。この他に高負荷時の場合を想定した測定もおこなった。200 クライアントから、毎秒計 20,000 の問合せで平均 1ms、最悪値で 6ms の応答速度であった。

Table 2: Redis の読み取り速度 (1 クライアント)

試行回数	1,000,000
平均	0.26ms
標準偏差	0.14ms

Table 3 に Cassandra の読み込み速度の測定結果を示す。Cassandra の 1 日分のデーター読み込み速度は約 1 秒であるがこれを多数のプロセスで分散読み込みをさせることで 0.2 秒程度にまで短縮されられることがわかった.

Table 3: Cassandra の読み取り速度 (1 クライアント)

試行回数	10,000
対象データー数	86,400 (1 日分)
平均	1.01s
標準偏差	0.18s

5. 結論

MADOCA のシステムの設計当時から最近までデーター管理システムでは RDBMS が主流であった。しかし近来の NoSQL 技術の進展により、分散化並列化された高性能、高信頼のシステムが低コストで得られるようになった。また分散コンピューティングのためのミドルウェアについても複雑化 (例えば CORBA) をたどっていたがより単純で高速なメッセージングが利用できるようになった。

今回これらの最新の技術を使用することによっていままでの MADOCA ではできなかった柔軟で、管理が容易かつ高性能で拡張性のあるシステムを実現できた。このシステムは 2013 年度のさらなる試験を経て 2014 年度に本格導入される予定である。これらを利用することにより加速器機器の管理がより容易にかつ精密におこなわれることが期待される。

参考文献

[1] R.Tanaka, et al., “The first operation of control system at the SPring-8 storage ring”, Proceedings of ICALEPCS 1997, Beijing, China, (1997) p.1.

[2] A.Yamahsita, et al., “The database system for the SPring-8 storage ring control”, Proceedings of ICALEPCS 1997, Beijing, China, (1997) p.427.

[3] Pieter Hintjens, “Zeromq Messaging for Many Applications”,O’Reilly Media,(2013).

[4] C.Fay, et al., “Bigtable: A Distributed Storage System for Structured Data”, Research, Google (2006).

[5] http://cassandra.apache.org/.

[6] Nancy Lynch and Seth Gilbert, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”, ACM SIGACT News, Volume 33 Issue 2 (2002), pg. 51-59.

[7] http://redis.io.

[8] http://memcached.org.

[9] 古橋貞之 “分散システムのためのメッセージ表現手法に関する研究”, 筑波大学大学院博士課程システム情報工学研究科修士論文, (2012).

[10] http://www.datastax.com/dev/blog/advanced-time-series-with-Cassandra.