Source型コンポーネント開発マニュアル

仲吉一男

June 2009

概 要

この文書では、Source 型コンポーネント開発方法について説明し ます。Source 型コンポーネントとは、リードアウトモジュール等のデータソースへアクセスしてデータを取得し後段のコンポーネントへ送るタイプのもので、DAQ コンポーネント間を流れるデータストリームの起点となるコンポーネントです。J-PARC MLF 中性子実験で使用している Gatherer コンポーネントがそれにあたります。PSD 検出器用の Gatherer コンポーネントは、NEUNET モジュール（PSD 検出器用リードアウト・モジュール）から複数のイベントデータを取得してヘッダとフッタをつけてデータブロックとして、後段の DAQ コンポーネントへ送信します。ここでは具体例として Echo サーバへ接続してデータを取得する EchoReader コンポーネントを作成します。

目 次

1 はじめに

DAQ ミドルウエアは、ネットワーク分散環境でデータ収集用ソフトウェアを容易に構築するためのソフトウェア・フレームワークです。ユーザは、DAQ コンポーネントと呼ばれるソフトウェア・コンポーネンを組み合わせて DAQ システムを構築することができます。DAQ コンポーネントは、コンポーネント間を流れるデータストリームへの関わり方で、いくつかのタイプに分類することができます。その中のひとつとして Source 型があります。Source 型の DAQ コンポーネントは、データストリームの起点となるコンポーネントで、例えばリードアウト・モジュールからデータ読み、後段のコンポーネントへ送るものです。別の言い方をすると InPort(データ入力ポート) を持たず、OutPort(データ出力ポート) を持つタイプの DAQ コンポーネントです。J-PARC MLF 中性子の PSD(Position Sensitive Detector) 検出器 DAQ システムで使用中の NEUNET モジュール（PSD 検出器用リードアウト・モジュール）からデータを取得する Gatherer コンポーネント (以下、Gatherer) は、Source 型のコンポーネントです。この文書では、まず Gatherer について説明をします。次に実際に Echo サーバへ接続してデータを取得する EchoReader コンポーネントを作成します。Echo サーバについては後述します。

2 Source 型コンポーネントの実例

前述の MLF 中性子で使用されている Gatherer について説明します。

2.1 MLF 中性子 PSD 検出器 DAQ システムの前提条件

MLF 中性子 PSD 検出器 DAQ システムでは次のような条件を満たす必要がありました。

- NEUNET から取得したデータはモジュール毎にファイルにして保存すること（要請）
- Gatherer は VME クレート 1 台中の NEUNET（最大 20 台）からデータを取得する。そのため DAQ コンポーネント間のデータストリーム上には最大 20 台分の NEUNET のデータが流れる
- NEUNET から一度に取得できるデータのサイズは最大 32kByte(8byte/event)

PSD 検出器用の Gatherer の仕様として、次の 2 点が挙げられます。

- 複数の NEUNET から複数のイベントデータを取得
- 上記前提条件を満たすため、取得したデータに対しモジュール毎のデータブロックにヘッダ、フッタをつけ、後段の DAQ コンポーネントへ送信する

2.2 Gatherer クラス

この PSD 検出器用の Gatherer のクラスを図 1 に示します。

すべての DAQ コンポーネントは図 1 の DaqComponentBase クラスを継承して実装します。DaqComponentBase クラスには、すべての DAQ コンポーネントの機能として共通な、状態遷移の実装、コマンド受信/ステータス送信機能等の実装があります。MlfComponent クラスには、MLF PSD 検出器に共通な機能が実装されています。その中にイベントデータのブロックへ付加するヘッダ、フッタデータの実装があり

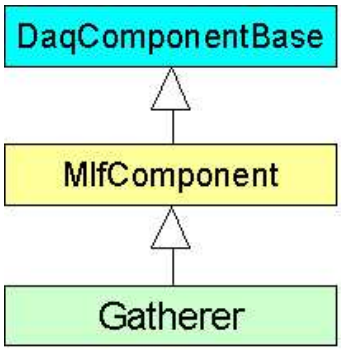


図 1:

ます。前述のように MLF PSD 検出器 DAQ システムでは、リードアウト・モジュール毎に取得したデータをファイルへ保存することが要請されており、そのファイル名は DAQ ID と呼ばれる DAQ 用計算機の ID と NEUNET のモジュール番号から生成することが決まっています。Gatherer はデータのヘッダに DAQ ID, モジュール番号、イベントデータのサイズ、フッタにはシーケンス番号等を付加しています。このシーケンス番号によりデータブロックがデータストリーム終点のコンポーネントまで正しく転送されているか確認しています。ヘッダ、フッタの詳細は後述します。

2.3 Gatherer の動作

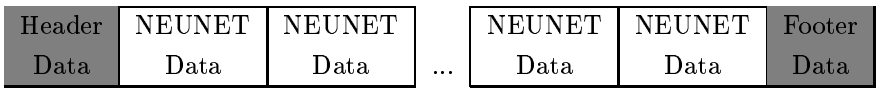
NEUNET からデータを取得する Gatherer は、DAQ オペレータからのコマンドにより次の動作を行います。( ) 内は J-PARC MLF 中性子で用いられているコマンド名です。

1. "Configure(Params)"コマンド投入時に DAQ オペレータからデータ取得を行う NEUNET の IP アドレスを DAQ オペレータから取得
2. "Start(Begin)"コマンドで上記 NEUNET へ接続
3. 動作中は NEUNET のデータ取得プロトコルに従い、socket を使用してデータを取得。
   取得したイベントデータに対しモジュール毎にヘッダ情報とフッタ情報をつけ、OutPort からデータを送信を繰り返す。
4. "Stop(End)"コマンドで NEUNET との接続を切断

2.4 データストリームのフォーマット

DAQ コンポーネント間を流れるデータストリームのフォーマットを示します。1 モジュール分のデータブロックとして複数（最大 4000 イベント）の NEUNET データの前後にヘッダとフッタデータがあります。各コンポーネントはフッタのシーケンス番号によりデータの消失がないかチェックを行い、あった場合には Fatal エラーとなります。データを保存する Logger コンポーネントは、ヘッダ情報から DAQ ID, モジュール番号を取得して保存するファイルを決定します。保存する際には、ヘッダとフッタ情報は取り除きます。

オンラインモニタを行う Monitor コンポーネントは、DAQ ID, モジュール番号から生成するヒストグラムファイル名を生成しています。



Gatherer は次のヘッダ、フッタ情報を NEUNET から取得したデータブロックに付加します。
ヘッダデータは、ヘッダマジック 2byte, DAQ ID 1byte, モジュール番号 1byte, データサイズ (ヘッダ、フッタ除く)4byte から構成されています。

表 1: ヘッダデータ

Header Magic (0xe7)	Header Magic (0xe7)	DAQ ID	Module NO.	Event Byte Size (24:31)	Event Byte Size (16:23)	Event Byte Size (8:15)	Event Byte Size (0:7)
0 7	8 15	16 23	24 31	32 39	40 47	48 55	56 63

表 2: フッタデータ

Footer Magic (0xcc)	Footer Magic (0xcc)	IP address (8:15)	IP address (0:7)	Sequence Number (24:31)	Sequence Number (16:23)	Sequence Number (8:15)	Sequence Number (0:7)
0 7	8 15	16 23	24 31	32 39	40 47	48 55	56 63

フッタデータは、フッタマジック 2byte, モジュールの IP アドレス第 4 オクテット 2byte, シーケンス番号 4byte から構成されています。

3 EchoReader コンポーネント 実装の準備

前節では、現在 J-PARC MLF PSD 検出器系 DAQ システムで使用されている Gatherer を例に Source 型コンポーネントについて解説を行いました。本節ではより単純な Source 型コンポーネントとして Echo サーバへ接続してデータを取得する EchoReader コンポーネント (以下、EchoReader) を作成します。EchoReader は、自分自身で 100 個の正規分布に従うデータを生成し、Echo サーバへ送信します。その後、Echo サーバから同じデータを受信し、後段の DAQ コンポーネントへ送信します。Echo サーバ（ポート 7 番）とは、名前の通り受信したデータをそのまま返すサービスを提供するサーバです。後述しますが Linux では、xinetd を使用して、このサービスを実行することができます。

EchoReader 実装のおおまかな手順を下記に示します。

1. 開発環境の準備
   新しい DAQ コンポーネントを開発する場合は、既存のコンポーネント（例えば、DAQ ミドルウェア for MLF の Gatherer コンポーネントや Monitor コンポーネント）から機能の似ているものを改良する方法と Skeleton コンポーネントというコアロジックが実装されていない DAQ コンポーネントへ必

要や機能を追加し実装する方法があります。この例題では Skeleton コンポーネントを使って開発する方法で説明します。

2. DAQ コンポーネントの状態遷移時の動作、状態中の動作を実装する
   DAQ コンポーネントは、状態遷移時に 1 回呼ばれるメソッド、その状態中に繰り返し呼ばれるメソッドがあります。DAQ コンポーネントは、"LOADED", "CONFIGURED", "RUNNING", "PAUSED" という状態が存在しており、それぞれの状態中にコンポーネントが行う動作を実装します。またある状態から別の状態へ遷移する際に 1 度行う動作を実装します。

3. DAQ コンポーネントで致命的なエラーが起きた場合は、それを DAQ オペレータへ報告します。詳細は後述します。

3.1 開発環境とそのインスト ール方法

この文章を作成する際に使用した開発環境は次の通りです。

- ハードウェア
  - ThinkPad X61(Windows XP Professional SP3)
- ソフトウェア
  - VMware Player 2.0.4
    - OS: Scientific Linux SL release 5.2 (Boron)
    - DAQ ミドルウェア: 2009.7 月版

開発環境のインストール等に関しては千代氏による付録 A を参照してください。

3.2 EchoReader 関連ファイルの作成と編集

付録 A の手順でインストール終了後、EchoReader 関連ファイルの作成と編集を行います。

`/home/daq/DaqComponents/src` の中にある Skeleton コンポーネントを利用して、EchoReader コンポーネントを作成します。Skeleton コンポーネントは、例題用コンポーネントです。コア・ロジック（コンポーネントの機能ロジック）は実装されていません。新しいコンポーネントを開発する際に利用できます。下記の 4 つのファイルを使用します。

- Skeleton.h
- Skeleton.cpp
- SkeletonComp.cpp
- Makefile.Skeleton

前述の 4 つの Skeleton 関連ファイルを次のような名前のファイルへコピーします。

- Skeleton.h → EchoReader.h

- Skeleton.cpp → EchoReader.cpp
- SkeletonComp.cpp → EchoReaderComp.cpp
- Makefile.Skeleton → Makefile.EchoReader

```
$ cp Skeleton.h EchoReader.h
$ cp Skeleton.cpp EchoReader.cpp
$ cp SkeletonComp.cpp EchoReaderComp.cpp
$ cp Makefile.Skeleton Makefile.EchoReader
```

下記のように名前を変更したファイル中に含まれている”Skeleton”,”skeleton”という文字列を”EchoReader”,”echoReader”という文字列に置換します。下記のコマンドは 1 行で入力してください。

```
$ for i in *EchoReader*; do
   sed -i.bak -e 's/Skeleton/EchoReader/g'
              -e 's/skeleton/echoReader/g'
              -e 's/SKELETON/ECHOREADER/g'  $i;
   done
```

Source 型コンポーネントはデータ入力ポートは使用しません。EchoReader.h の InPort 関連の行をコメントアウトします。

```
private:
    //TimedOctetSeq m_in_data;
    //InPort<TimedOctetSeq, MyRingBuffer> m_InPort;

    TimedOctetSeq m_out_data;
    OutPort<TimedOctetSeq, MyRingBuffer> m_OutPort;
```

EchoReader.cpp のコンストラクタを次のように変更します。入力ポート関連の行をコメントアウトします。

```
EchoReader::EchoReader(RTC::Manager* manager)
    : DAQMW::MlfComponent(manager),
      ///m_InPort("echoReader_in",   m_in_data),
      m_OutPort("echoReader_out", m_out_data),

      ///m_in_status(BUF_SUCCESS),
      m_out_status(BUF_SUCCESS),

      m_debug(false)
{
    // Registration: InPort/OutPort/Service

    // Set InPort buffers
    ///registerInPort ("echoReader_in",   m_InPort);
    registerOutPort("echoReader_out", m_OutPort);

    init_command_port();
    init_state_table( );
    set_comp_name(COMP_NONAME);
}
```

3.3 EchoReader のビルドと基本動作確認

付録 B にあるファイルを使用して、EchoReader のみのテストを行う場合は、下記のように EchoReader.cpp の冒頭で `#define USE_OUTPORT` をコメントアウトしてください。コメントアウトするとデータを OutPort から送信しません。EchoMonitor と接続して使用する際は、コメントアウトせずに make してください。

- EchoReader だけでテストする場合（OutPort からデータ送信なし）

```
#include "EchoReader.h"

/// For stand alone test, comment out next line.
//#define USE_OUTPORT
```

- EchoReader と EchoMonitor を接続してテストする場合（OutPort からデータ送信あり）

```
#include "EchoReader.h"

/// For stand alone test, comment out next line.
#define USE_OUTPORT
```

下記のように make を行い、コンポーネントがビルドできるか確認します。

```
$ cd your_directory/DaqComponents
$ make -f Makefile.EchoReader
```

コンパイルでエラーが出た場合は、原因を調べて修正を行ってください。問題なくコンパイルが終了したら、EchoReader コンポーネントを開発するための準備が整ったことになります。ビルドが成功したら、次の手順でコンポーネントを起動してコマンドを送って動作を確認します。まだ Echo サーバへ接続してデータを送信/受信するためのメインロジックの実装は行っていないので、DAQ オペレータからコマンドを受信して状態が遷移できるか確認します。動作が確認できたら、DAQ コンポーネントとしての基本機能は実装できたことになります。その後、Echo サーバへ接続してデータを送信/受信するためのメインロジックの実装を行います。テスト用に次のコンフィグレーション・ファイルを使用します。この例では、PC の IP アドレスとしてローカル・ループバック・アドレスの 127.0.0.1 を使用します。これは外部のネットワークへ接続する必要がない場合で、実際の実験等での使用では各自のネットワーク環境に合わせて IP アドレスを指定します。また下記の”/home/daq/DaqComponents” は、各自の環境に合わせて”bin/EchoReaderComp” がある場所への絶対パスを書いてください。

```
<?xml version="1.0"?>
<configInfo>
   <daqOperator>
      <hostAddr>127.0.0.1</hostAddr>
   </daqOperator>
   <daqGroups>
      <daqGroup gid="group0">
         <components>
            <component cid="EchoReader0">
               <hostAddr>127.0.0.1</hostAddr>
               <hostPort>50000</hostPort>
               <instName>EchoReader0.rtc</instName>
               <execPath>/home/daq/DaqComponents/bin/EchoReaderComp</execPath>
               <confFile>/home/daq/DaqComponents/rtc.conf</confFile>
               <startOrd>1</startOrd>
               <inPorts>
               </inPorts>
               <outPorts>
               </outPorts>
               <params>
               </params>
            </component>
         </components>
      </daqGroup>
   </daqGroups>
</configInfo>
```

上記 XML ドキュメントを例えば（名前は任意）config-echo.xml という名前で config.xml があるディレクトリに保存します。その後、”run-local.py” というテスト用の起動スクリプトを使用して下記のように起動します。”run-local.py” は、ローカル計算機上の DAQ コンポーネントを使用する場合に使用します。

```
$ ./run-local.py -c -f config-echo.xml
```

上記スクリプトのコマンドライン・オプション ”-c” は、コンソールからのコマンド入力モードの選択、”-f” はコンフィグレーション・ファイルのパスの指定に使用します。

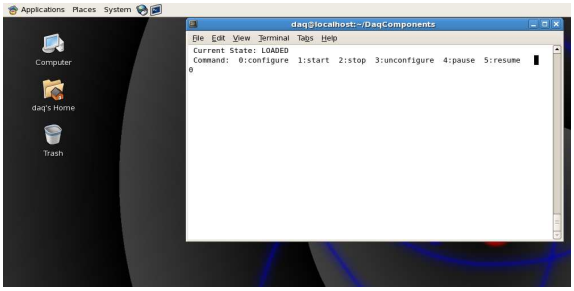


図 2: スクリプト実行後の DAQ オペレータのコマンド待ち受け画面

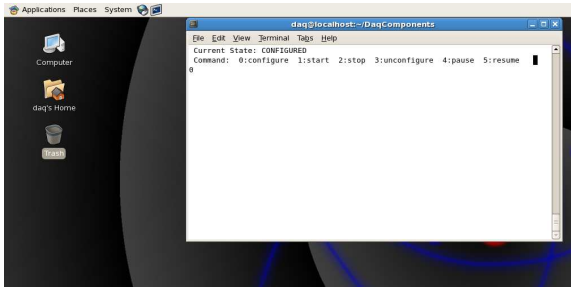


図 3: EchoReader Configure 後の DAQ オペレータの画面

run-local.py スクリプトは、先ほど作った config-echo.xml を読み込んで、EchoReader コンポーネントを起動します。その後、DAQ オペレータを起動します。DAQ オペレータは、自身のサービスポートと EchoReader のサービスポートを接続し、コマンド/ステータス通信経路を確立します。

図 2 に、run-local.py 実行後のスクリーンショットを示します。スクリーンショット中の右側 GNOME 端末のの上部に `Current State: LOADED` があります。これは、EchoReader が `LOADED` 状態であることを示しています。その下にコマンドが表示されています。コマンドに対応した数字を入力してリターンキーを押してください。”0”を入力して `Configure` コマンドを実行してください。EchoReader でコマンドが実行されると図 3 のように `Current State: CONFIGURED` になります。画面上のコマンドに対応した番号を入力してリターンキーを押すことで、DAQ オペレータから EchoReader コンポーネントへコマンドを送り、状態を遷移させることができることを確認してください。`1:start` コマンド入力後にラン番号を入力のプロンプトが出ますが、この例題ではラン番号は使用しないので適当な数字を入れてください。

4 EchoReader の動作の実装

EchoReader の動作確認が終了したら、Echo サーバへアクセスしてデータを取得するための実装を行います。図 4 に状態遷移図とその際に呼ばれるメンバ関数を示します。青い文字のメンバ関数は遷移時に 1 度呼ばれるもので、赤い文字のものはその状態中に繰り返し呼ばれます。これらのメンバ関数を実装することで色々な機能の DAQ コンポーネントを実現することができます。

これから次の手順で EchoReader に実装を行います。

- Makefile.EchoReader への追加
- EchoReader.h への追加
- EchoReader.cpp への追加

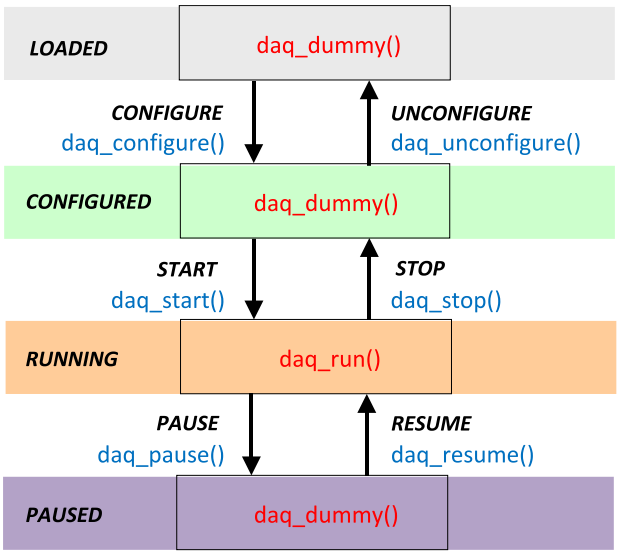


図 4: DAQ コンポーネントの状態遷移と関連するメンバ関数

- 上記の各状態および状態遷移時に呼ばれるメンバ関数の実装

4.1 Makefile.EchoReader の編集

Makefile.EchoReader へ下記の 1 行め SOCKINC, 2 行め CXXFLAGS, 11 行め SOCKLIB のような追加・変更します。

```
SOCKINC  = -I../../SiTCP/CPP/Sock
CXXFLAGS = `rtm-config --cflags` $(SOCKINC)
LDFLAGS  = `rtm-config --libs`
SHFLAGS  = -shared

IDLC     = `rtm-config --idlc`
IDLFLAGS = `rtm-config --idlflags` -I`rtm-config --prefix`/include/rtm/idl
WRAPPER  = rtm-skelwrapper
WRAPPER_FLAGS = --include-dir="" --skel-suffix=Skel --stub-suffix=Stub

SOCKLIB = -L../../lib -lSock
```

下記の 2 行めのように$(SOCKLIB) を追加します。

```
EchoReaderComp: EchoReaderComp.o $(OBJS)
        $(CXX) -o $@ $(OBJS) EchoReaderComp.o $(LDFLAGS) $(SOCKLIB)
        cp $@ $(BINDIR)/$@
```

4.2 EchoReader.h の編集

- EchoReader.h の変更。
  DAQ ミドルウェアで使用している Socket クラスの Sock.h をインクルードします。これは、Echo サーバへ接続してデータを取得する際に使用します。また、EchoReader.h に次のメンバ関数、変数を追加します。

```
...
// Service implementation headers
#include "DAQServiceSVC_impl.h"

#include "Sock.h"    //Socket wrapper class

using namespace RTC;
...
```

```
class EchoReader
    : public DAQMW::MlfComponent
{
...
private:
    int daq_dummy();
    ...
    int daq_resume();

    int parse_params(::NVList* list);

    unsinged int getGauss();                              //gauss 分布データ取得用関数    追加
    int send_data_to_echoServer();                        //Echo サーバへデータ送信関数    追加
    int recv_data_from_echoServer();                      //Echo サーバからデータ受信関数 追加
    int set_data_to_OutPortBuf(unsigned int seq_num); //送信用バッファへデータセット  追加
    int check_outPort_status(int status);                 //送信ステータスチェック関数    追加

    //int m_in_status;
    int m_out_status;

    DAQMW::Sock* m_sock;                                  //Sock ポインタ変数 追加
    int m_dataByteSize;                                   //データのバイト数   追加
    unsigned char* m_wdata;                               //書き込みデータ用バッファ 追加
    unsigned char* m_rdata;                               //読み込みデータ用バッファ 追加

    bool m_debug;

    static const std::string HOSTADDR;                   //ホストの IP アドレス 追加
    static const int PORTNO = 7;                          //Echo サーバのポート番号 追加
};
```

4.3 EchoReader.cpp の編集

- コンストラクタの編集
  下記のように HOSTADDR, EVENT_BYTE_SIZE を追加し、コンストラクタを次のように変更します。

```
const std::string EchoReader::HOSTADDR = "127.0.0.1"; ///localhost
const unsigned int EVENT_BYTE_SIZE = 4;

EchoReader::EchoReader(RTC::Manager* manager)
    : DAQMW::MlfComponent(manager),
      ///m_InPort("echoReader_in",   m_in_data),
      m_OutPort("echoReader_out", m_out_data),

      ///m_in_status(BUF_SUCCESS),
```

```
      m_out_status(BUF_SUCCESS),
      m_sock(0),                              //Sock の初期化
      m_dataByteSize(400),                    //データバイト数を 400byte(100event) 初期化
      m_wdata(0),                             //書き込みバッファ・ポインタの初期化
      m_rdata(0),                             //読み込みバッファ・ポインタの初期化
      m_debug(false)
{
    // Registration: InPort/OutPort/Service

    // Set InPort buffers
    ///registerInPort ("echoReader_in",   m_InPort);
    registerOutPort("echoReader_out", m_OutPort);

    init_command_port();
    init_state_table( );
    set_comp_name(COMP_NONAME);

    m_eventByteSize = EVENT_BYTE_SIZE;
}
```

- デストラクタの編集
  デストラクタを次のように変更します。

```
EchoReader::~EchoReader()
{
    if(m_wdata) {
    delete [] m_wdata;
    m_wdata = 0;
    }
    if(m_rdata) {
    delete [] m_rdata;
    m_rdata = 0;
    }
}
```

- getGauss() の追加
  一様乱数を利用して、正規分布データを生成します。区間 [0, 1) の実数型一様乱数の和は、足し合わせる和の数が増えるにつれ正規分布に近づくことが知られています。特に 12 個の和は、平均 6、標準偏差 1 になります。次の関数では、得られた値を 10 倍しているので、平均値が 60、標準偏差が 10 の分布が得られます。

```
unsigned int EchoReader::getGauss()
{
    double gdat = 0.0;
    double amp  = 10.0;
    int loop = 12; /// distributin for mean = 6, sigma = 1
    for (int i = 0; i< loop; i++) {
        gdat = gdat + (double)random()/((double)RAND_MAX + 1);
    }
    return (unsigned)(gdat * amp);
}
```

- send_data_to_echoServer() の追加
  send_data_to_echoServer() は、自身で生成した正規分布のデータ 100 イベントを Echo サーバへ送信する関数です。

  1. getGauss() を 100 回呼んで、400byte のデータを配列 m_wdata へ詰めます。
     2 バイト以上の整数データをネットワークで転送する際は、htonl() を使用してネットワーク・

バイトオーダに変換する必要がありますが、この例ではローカルホスト上の echo サーバへの転送のため、この処理は省略します。

2. m_sock->writeAll() で echo サーバへデータを送信します。
   writeAll() は、指定したバイト数のデータをすべて転送するまで動作する関数です。
3. writeAll() の戻り値をチェックします。
   DAQMW::Sock::ERROR_FATAL および DAQMW::Sock::ERROR_TIMEOUT はそれぞれ Fatal エラー、タイムアウトが発生したことを示します。それぞれ USER_ERROR1,USER_ERROR2 という Fatal エラータイプと戻り値を指定して fatal_error_report() を呼びます。これにより、Fatal エラーが DAQ オペレータへ通知され EchoReader はアイドル状態になります。現在の実装では、Fatal エラーのタイプは、DAQService.idl 中で enum CompFatalTypes として定義されているため、ユーザが自由に Fatal エラーを追加することができません。今後この仕様を改善する予定です。

```
int EchoReader::send_data_to_echoServer()
{
    for (int i = 0; i < m_dataByteSize; i+=4) {
        unsigned int gdat = getGauss();
        //*(unsigned*)&m_wdata[i] = htonl(gdat);
        *(unsigned*)&m_wdata[i] = gdat;
    }
    int status = m_sock->writeAll(m_wdata, m_dataByteSize);
    if(status == DAQMW::Sock::ERROR_FATAL) {
        std::cerr << "fatal error..." << std::endl;
        fatal_error_report(USER_ERROR1, -1);
    }
    else if(status == DAQMW::Sock::ERROR_TIMEOUT) {
        std::cerr << "Timeout.. retry..." << std::endl;
        fatal_error_report(USER_ERROR2, -2);
    }
    return 0;
}
```

- recv_data_from_echoServer() の追加

  recv_data_from_echoServer() は、Echo サーバからデータを受信する関数です。readAll() は、指定したバイト数を読み込むまで動作する関数です。送信と同様に 2 バイト以上の整数データの受信の場合は、ntohl() 等でネットワーク・バイトオーダから host のバイトオーダへ変換する必要がありますが、前述の理由により省略しています。

```
int EchoReader::recv_data_from_echoServer()
{
    int status = m_sock->readAll(m_rdata, m_dataByteSize);
    if(status == DAQMW::Sock::ERROR_FATAL) {
        std::cerr << "fatal error..." << std::endl;
        fatal_error_report(USER_ERROR1, -3);
    }
    else if(status == DAQMW::Sock::ERROR_TIMEOUT) {
        std::cerr << "Timeout.. retry..." << std::endl;
        fatal_error_report(USER_ERROR2, -4);
    }
    unsigned char* rdata_ptr = &m_rdata[0];
    for (int i = 0; i < m_dataByteSize; i++) {
        unsigned int rdat = *(unsigned int*)rdata_ptr;
        //std::cerr << ntohl(rdat) << std::endl;
        std::cerr << rdat << std::endl;
        rdata_ptr += 4;
    }
    return 0;
}
```

- set_data_to_OutPortBuf() の追加
  set_data_to_OutPortBuf() は、OutPort のバッファにデータをコピーするための関数です。ここで使用しているヘッダ、フッタ情報は、J-PARC MLF 中性子で使用してものです。簡単のため IP アドレス情報 (addr), DAQ ID 値 (daqId), モジュール番号 (modNo) は 0 にします。ヘッダ、フッタに値をセットし、OutPort のバッファ・サイズをセットしてヘッダデータ、データ、フッタデータをコピーします。

```
int EchoReader::set_data_to_OutPortBuf( unsigned int seq_num )
{
    unsigned int addr  = 0;
    unsigned int daqId = 0;
    unsigned int modNo = 0;

    unsigned char header[8];
    unsigned char footer[8];

    set_header(header, daqId, modNo, m_dataByteSize);
    set_footer(footer, addr, seq_num);

    ///set OutPort buffer length
    m_out_data.data.length(m_dataByteSize + HEADER_BYTE_SIZE + FOOTER_BYTE_SIZE);

    memcpy( &(m_out_data.data[0]), header, HEADER_BYTE_SIZE);
    memcpy( &(m_out_data.data[HEADER_BYTE_SIZE]), &m_rdata[0], m_dataByteSize);
    memcpy( &(m_out_data.data[HEADER_BYTE_SIZE + m_dataByteSize]), footer, FOOTER_BYTE_SIZE);

    return 0;
}
```

- check_outPort_status() の追加
  check_outPort_status() は、OutPort からデータ送信した際のステータスをチェックする関数です。転送後のステータスをチェックして、BUF_SUCCESS なら m_loop というシーケンス番号をインクリメントし、送信したイベント数をインクリメントします。BUF_FATAL なら "-1" を返します。"-1"を受け取った EchoReader::daq_run() では Fatal エラーとなります。

```
int EchoReader::check_outPort_status(int status)
{
    int ret = 0;

    if (status == BUF_SUCCESS) {
        m_loop++;
        m_total_event += (m_dataByteSize/m_eventByteSize);
        if (m_debug) {
            if (m_loop%100 == 0) {
                std::cerr << "EchoReader: m_loop = " << m_loop << std::endl;
                std::cerr << "\033[A\r";
            }
        }
    }
    else if (status == BUF_TIMEOUT) {
        std::cerr << "EchoReader::Time Out occurred..." << std::endl;
    }
    else if (status == BUF_FATAL) {
        std::cerr << "### EchoReader::Fatal error occurred..." << std::endl;
        ret = -1;
    }
    return ret;
}
```

4.4 各状態と状態遷移の実装

4.4.1 LOADED 状態の実装

LOADED 状態は、DAQ コンポーネントが起動した後の状態です。daq_dummy() が繰り返し実行されています。

```
int EchoReader::daq_dummy()
{
    return 0;
}
```

現在の実装では、EchoReader::daq_dummy() は DaqComponentBase::daq_base_dummy() から呼ばれています。daq_base_dummy() では、下記のように CPU を消費しないように sleep() を呼び、アイドル状態となっています。各コンポーネントは、自身の daq_dummy() により独自の動作を追加できます。

```
  int DaqComponentBase::daq_base_dummy()
  {
      daq_dummy();
      set_status(COMP_WORKING);
      sleep(1);
      return 0;
  }
```

4.4.2 LOADED 状態から CONFIGURE 状態遷移時の実装

DAQ コンポーネントが LOADED 状態中に”Configure”コマンドを受信した場合、EchoReader::daq_configure() が 1 度実行されます。DAQ オペレータから送信されるパラメータ・リストをパーズしてパラメータを抽出し、設定を行います。この例題では、パラメータの受信は行いません。Echo サーバへのデータ送受信に使用する配列を確保します。このデータ送受信用配列の確保するタイミングについては case-by-case で判断してください。その後、CONFIGURED 状態へ遷移します。

```
int EchoReader::daq_configure()
{
    std::cerr << "*** EchoReader::configure" << std::endl;

    ::NVList* paramList;
    paramList = m_daq_service0.getCompParams();
    parse_params(paramList);

    m_rdata = new unsigned char[m_dataByteSize];
    m_wdata = new unsigned char[m_dataByteSize];
    return 0;
}
```

4.4.3 CONFIGURED 状態の実装

CONFIGURED 状態は、LOADED 状態と同様の EchoReader::daq_dummy() が呼ばれます。

4.4.4 CONFIGURED 状態から RUNNING 状態遷移時の実装

CONFIGURED 状態中に”Start”コマンドを受信した場合、RUNNING 状態へ遷移する前に必要な処理を行います。EchoReader::daq_start() が 1 回実行されます。

EchoReader の場合は、EchoServer との接続を確立します。その後、RUNNING 状態へ遷移します。

```
int EchoReader::daq_start()
{
    std::cerr << "*** EchoReader::start" << std::endl;

    ///m_in_status  = BUF_SUCCESS;
    m_out_status = BUF_SUCCESS;

    try {
      m_sock = new DAQMW::Sock(); ///
      m_sock->connect(HOSTADDR, PORTNO);   ///
    } catch (DAQMW::SockException& e) {
        std::cerr << "Sock Fatal Error : " << e.what() << std::endl;
    } catch (...) {
        std::cerr << "Sock Fatal Error : Unknown" << std::endl;
    }

    return 0;
}
```

4.4.5 RUNNING 状態の実装

EchoReader の RUNNING 状態の動作は次のようになります。

1. 自身で正規分布のデータ 100 イベントを生成して、Echo サーバへ送信する
2. Echo サーバからデータを受信する（前のシーケンスで送信したデータがエコーバックされる）
3. 受信したデータブロックにヘッダ、フッタを付加して、OutPort の送信用バッファへコピーする
4. OutPort からデータを送信する
5. 送信ステータスをチェックし、Fatal エラーの場合は報告する

上記のシーケンスの実装を下記に示します。RUNNING 状態では、下記の `daq_run()` が繰り返し実行されます。処理の最後に usleep() で 100ms スリープしていますが、これは EchoMonitor と接続してテストを行う際に、使用するの計算機の処理能力によらず、一定時間でヒストグラムをアップデートするために行っています。実際に実験等で使用する Source 型のコンポーネントでは必要ありません。

```
int EchoReader::daq_run()
{
    send_data_to_echoServer();
    recv_data_from_echoServer();

    set_data_to_OutPortBuf(m_loop);
    m_out_status = m_OutPort.write(m_out_data); /// send data to next component

    if (check_outPort_status(m_out_status) == -1) {
        std::cerr << "### EchoReader: OutPort.write(): FATAL ERROR\n";
        fatal_error_report(USER_ERROR3, -1);
        return 0;
    }

    if (check_trans_lock() ) {  /// got stop command
        set_trans_unlock();
        return 0;
    }

    usleep(100000); /// sleep for 100ms to adjust histogram update time
    return 0;
}
```

DAQ コンポーネントは、daq_run() 中で、下記のようにストップコマンドを受信したかチェックを行っています。ストップコマンドを受信すると、RUNNING 状態から CONFIGURED 状態へすぐに遷移するのではなく、遷移がロックされます。これは、例えばデータ転送を行っている最中に”Stop”コマンドが非同期でくる可能性があるからです。DAQ コンポーネントは、daq_run() の動作シーケンス中で遷移可能なポイントに達したら、 set_trans_unlock() を呼んで次の状態へ遷移できるようにします。

```
    if (check_trans_lock() ) {  /// got stop command
        set_trans_unlock();
        return 0;
    }
```

4.4.6 RUNNING 状態から PAUSED 状態遷移時の実装

RUNNING 状態中に”Pause”コマンドを受信すると EchoReader::daq_pause() が 1 度実行され PAUSED 状態へ遷移します。その際、この例では何も行っていません。

```
int EchoReader::daq_pause()
{
    std::cerr << "*** EchoReader::pause" << std::endl;
    return 0;
}
```

PAUSED 状態中、Echo サーバとの TCP 接続を切断する場合と維持する場合が考えられます。EchoReader では、接続を維持しています。

4.4.7 PAUSED 状態の実装

PAUSED 状態では、EchoReader::daq_dummy() が呼ばれアイドル状態となります。

4.4.8 PAUSED 状態から RUNNING 状態遷移時の実装

PAUSED 状態中に”Resume”コマンドを受信すると EchoReader::daq_resume() が 1 度実行され RUNNING 状態へ遷移します。この例では、何も行っていません。

```
int EchoReader::daq_resume()
{
    std::cerr << "*** EchoReader::resume" << std::endl;
    return 0;
}
```

5 致命的エラー処理

致命的エラーとは、そのエラー後は DAQ コンポーネントが正常に動作を行えなくなる場合です。この場合、コンポーネントは fatal_error_report() というメンバ関数を呼んで DAQ オペレータへ Fatal エラーを報告し自身はアイドル状態になります。報告を受けた DAQ オペレータは、それを上位システム、またはユーザへ通知し判断を待ちます。通常はストップコマンドによるランの終了や強制終了によるシステムの再立ち上げを行います。例えば、上記 daq_run() 中で、OutPort からデータを送信後、そのステータスを確認しています。

```
    if (check_outPort_status(m_out_status) == -1) {
        std::cerr << "### EchoReader: OutPort.write(): FATAL ERROR\n";
        fatal_error_report(USER_ERROR3, -1);
        return 0;
    }
```

6 さいごに

Source 型 DAQ コンポーネントの例として MLF 中性子で使用中の Gatherer の説明をしました。DAQ-Middleware for MLF の中に入っている Skeleton コンポーネントを変更して、Echo サーバからデータを取得する EchoReader コンポーネントの開発の手順を説明しました。完成した EchoReader をテストするためには、EchoReader からデータを受信する DAQ コンポーネントが必要です。「Sink 型コンポーネント開発マニュアル」[1] には、例題として EchoReader からデータを受信してヒストグラムにして表示を行う EchoMonitor コンポーネントがあります。EchoReader と EchoMonitor を作成し両者を接続して最終的なテストを行ってください。

参考文献

[1] 千代浩司、Sink 型コンポーネント開発マニュアル、2009 年 7 月。
http://greeetea.kek.jp/daqm/docs/sink-comp.pdf

A　コンポーネント開発環境の準備

A.1　vmplayer の利用

この文書にそって開発する環境を用意するには vmplayer のイメージを取得し vmplayer 上で行うのが簡単です。vmplayer のセットアップ作業については `http://greentea.kek.jp/daqm/vmplayer/` をご覧ください。

vmplayer のイメージは `http://greentea.kek.jp/daqm/vmplayer/sl_53.zip` にあります。この vmplayer イメージにはこの文書でのべた作業を実行するのに必要な開発環境がはいっています。ROOT は `/usr/local/root/`以下にインストールされています。一般ユーザーとしてアカウント名 `daq`、パスワード `daqone` が登録されています。また `root` のパスワードは `abcd1234` です。パスワードは適切に変更してください。

この vmplayer イメージを使用したコンポーネント開発は`/home/daq/DaqComponents/`以下で行います。たとえば `EchoReader.h` などは`/home/daq/DaqComponents/src/` ディレクトリで作成します。この文書にそってコンポーネントを作る際に必要になるソースファイルおよびその変更の詳細については本書本文をお読みください。

A.2　自力で Linux 上にコンポーネント開発環境を準備する場合

上記 vmplayer を利用せず、新たに自力で Linux 上にこの文書のコンポーネントを開発する環境を準備する手順を以下に書きます。OS は RedHat Enterprise Linux (RHEL) 5.3 あるいは Scientific Linux (SL) 5.3 を想定します。コンポーネント動作には OpenRTM-aist KEK 版が必要です。OpenRTM-aist KEK 版バイナリ RPM は用意されていますが、RHEL 5.2、RHEL 5.3、SL 5.2、SL 5.3 以外ではテストされていません。これら以外の OS、Linux distribution で OpenRTM-aist KEK 版バイナリ RPM がそのまま動作するかどうかは不明です。

以下 RHEL 5.3 あるいは SL 5.3 上に環境を用意するとして解説を行います。

A.2.1　gcc, g++, make など

開発には gcc, g++, make など通常のソフトウェア開発で必要になるユーティリティが必要です。これらのセットアップについてはここでは解説しません。

A.2.2　ROOT

sink 型コンポーネントではヒストグラムを作るのに ROOT を使用しますので ROOT が必要です。`http://root.cern.ch/` を見て準備してください。ここでは ROOT のインストールについては解説しません。

A.2.3　OpenRTM-aist KEK 版

root ユーザになって以下のコマンドを実行します。

```
# rpm -ihv http://www-jlc.kek.jp/%7Esendai/OpenRTM/EL5/noarch/ (→)
kek-daqmiddleware-repo-1-3.el5.noarch.rpm
```

(1行が長いので (→) で改行していますがコマンドとしては1行で入力します) これで/etc/yum.repos.d/kek-daqmiddlewar
がインストールされます。このファイルは次で実行する yum の設定ファイルとして使われます。

次に

```
# yum --enablerepo=kek-daqmiddleware install OpenRTM-aist xerces-c-devel xalan-c-devel
```

とコマンドを投入します。途中で yes/no を聞いてきますので y を入力します。RPM パッケージが置いてあるサーバーから自動でダウンロードが実行され、以下の RPM パッケージがインストールされます。

- ACE
- ACE-devel
- omniORB
- omniORB-bootscripts
- omniORB-devel
- omniORB-doc
- omniORB-servers
- omniORB-utils
- xerces-c
- xerces-c-devel
- xalan-c
- xalan-c-devel
- OpenRTM-aist

この状態で計算機を再起動すると omniNames サーバーが自動で起動するようになっていますので、自動起動しないようにするために以下のコマンドを実行します。

```
# /sbin/chkconfig omniNames off
```

A.2.4 DAQ-Middleware ソースファイル

続いて DAQ-Middleware for MLF のソースファイル一式をダウンロードして展開します。以下ではユーザー daq で、ディレクトリは/home/daq 以下でコンポーネント開発を行うと仮定した場合のコマンドを示します。また shell として bash、あるいは zsh を使っていると仮定します。

```
daq としてログインする。
daq% cd /home/daq
daq% mkdir tars.2009.07
daq% cd tars.2009.07
daq% lftp http://www-jlc.kek.jp/~sendai/OpenRTM/EL5/tars.2009.07/
lftp> mget *.tar.gz
lftp> quit
```

```
daq% for i in *.tar.gz; do
tar xf $i -C ..
done
daq%
```

以上の作業で/home/daq/DaqComponents/等のディレクトリができます。この文書で行っているコンポーネント開発は/home/daq/DaqComponents/以下の各ディレクトリで実行します。

A.2.5 SELinux

ここで作成するコンポーネントはネットワークソケットを使うために libSock.so シェアードライブラリ (この文書のインストール例だと/home/daq/lib/libSock.so) を使用します。RHEL あるいは SL のデフォルトのインストール方法では SELinux が enforcing になっていて、この状態では libSock.so シェアードライブラリを使用することができません。各自の判断でこのシェアードライブラリを使えるように設定してください。設定例を以下に書きます。

SELinux を disabled にする

SELinux を disalbed にするには/etc/sysconfig/selinux ファイルで SELINUX=enforcing になっている行を SELINUX=disabled に変更します。変更後、再起動が必要です。

chcon -t を使う

```
root# chcon -t texrel_shlib_t /home/daq/lib/libSock.so
```

A.2.6 Echo サーバーの動作確認

続いて EchoReader コンポーネントの動作のために xinetd パッケージがインストールされているかどうかを確認します。

```
rpm -q xinetd
```

このコマンドでなにも出力されない場合は xinetd パッケージはインストールされていません。xinetd パッケージは RHEL、SL で提供されているパッケージなので OS 配布パッケージからインストールしてください。

Source コンポーネントの動作には Echo サーバーが動いていて Echo サーバーと通信できることが必要です。

通信できるかどうかの確認はたとえば telnet コマンドを使って

```
% telnet localhost 7
Hello, world
```

として行います。キー入力後、リターンキーを押した直後に入力した行がそのまま表示されれば Echo サーバーと通信できています。telnet コマンドから抜けるには Ctrl-] を押して telnet> プロンプトが出たところで close と入力します (GNOME ターミナルをお使いの場合は Ctrl-] を押したあとにリターンキーを押さないと telnet> プロンプトが出ないことがあります)。

通信できなかった場合は

- xinetd が動いているか
- Echo サーバーが動くようになっているか

確認します。xinetd が動いているかどうかは `pgrep -fl xinetd` で確認します。動いていれば

```
12345 xinetd -stayalive -pidfile /var/run/xinetd.pid
```

のように表示されます (先頭の数字はプロセス番号ですので必ずこの値になっているわけではありません)。動いていなければ`/etc/init.d/xinetd start` で起動してください。

Echo サーバーが動くようになっているかどうかの確認は`/etc/xinetd.d/echo-stream` ファイルを見て `disabled = no` になっているかどうかで確認します。yes になっていれば root ユーザーになって no に書き換えて`/etc/init.d/xinetd restart` で再起動しておきます。xinetd を自動起動するには root ユーザーで`/sbin/chkconfig xinetd on` とします。

A.2.7 MLF 向けコンポーネントのコンパイル、動作

MLF 向けコンポーネントをコンパイル、動作させる場合には以下のコマンドを実行します。この文書で開発するコンポーネントだけ動作すればよい場合は実行する必要はありません。

```
# yum --enablerepo=kek-daqmiddleware install gsl-devel mxml
```

B　ソースコードの入手方法

この文章で説明した EchoReader コンポーネント関連のファイルは `http://greentea.kek.jp/daqm/src/source-comp-2009.tar.gz` にあります。上記ファイルを展開すると echo-reader ディレクトリが作成され、その中に次のディレクトリとファイルが作成されます。その中の config-echo.xml は EchoReader 単独で動作させる際のものです。EchoMonitor と接続して動作させるためには、「Sink 型コンポーネント開発マニュアル」[1] で作成するコンフィグレーション・ファイルを使用してください。

```
echo-reader/
|-- config-echo.xml
`-- src
    |-- EchoReader.cpp
    |-- EchoReader.h
    |-- EchoReaderComp.cpp
    `-- Makefile.EchoReader
```