Sink 型コンポーネント開発マニュアル

高エネルギー加速器研究機構
素粒子原子核研究所
千代浩司

2009 年 7 月

概要

モニターコンポーネント、ロガーコンポーネントなど他のコンポーネントからデータを受け取り、他のコンポーネントへデータは送ることはしないコンポーネント (Sink 型コンポーネント) の開発方法について解説します。話を具体的にするために、まず「Source 型コンポーネント開発マニュアル」[3] で実装した EchoReader コンポーネントからデータを受け取るだけのコンポーネントを作成します。上流コンポーネントからのデータの読みだしができるようになれば、あとは通常のプログラミングでそのデータをディスクにセーブするようにすればロガーコンポーネントになりますし、ヒストグラムを書くようにすればモニターコンポーネントになります。この文書では、続けてデータを読むだけのコンポーネントを改造し ROOT でヒストグラムを画面上に書くモニターコンポーネントを開発します。開発環境の準備については付録にまとめてあります。また、この文書中で作成するソースファイルの入手方法についても付録をご覧ください。

目次

1 Sink 型コンポーネントについて 3

2 開発方法 3

3 DAQ-Middleware での状態と状態遷移、およびエラー処理 3

4 この文書で実装する Sink 型コンポーネントについて 4
- 4.1 上流データコンポーネント 4
- 4.2 データフォーマットの確認 5
- 4.3 実装準備 6

5 上流コンポーネントからのデータを読むコンポーネントの作成 7
- 5.1 EchoMonitor.h の変更 7
- 5.2 EchoMonitor.cpp の変更 8

5.3 上流コンポーネントからの読みだしテスト 12

6 ROOT を使ってヒストグラム図を画面に表示する 15
6.1 Makefile.EchoMonitor の変更 15
6.2 EchoMonitor.h の変更 16
6.3 EchoMonitor.cpp の変更 17
6.4 EchoMonitorComp.cpp の変更 18
6.5 make の実行 18
6.6 ヒストグラム表示テスト 19

7 パラメータの Condition データベース化 19
7.1 実装 19
7.2 Condition データベースを使ったヒストグラムのテスト 22

付録 A コンポーネント開発環境の準備 24
A.1 vmplayer の利用 24
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 24

付録 B ソースコードの入手法 28

1 Sink 型コンポーネントについて

DAQ-Middleware はネットワーク分散環境でデータ収集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。ユーザーは 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トを組み合わせて DAQ システムを構築することができます。

DAQ コンポーネントはそのデータ流の扱いかたによりいくつかのタイプに分類することができます。そのひとつとして Sink 型 DAQ コンポーネントがあります。Sink 型コンポーネントは上流 DAQ コンポーネントからデータを受けとりますが、他の DAQ コンポーネントにデータを送ることはしないタイプの DAQ コンポーネントで、たとえばデータを受け取ってディスクに書く DAQ コンポーネント (ロガー)、ヒストグラムを書く DAQ コンポーネント (モニター) がこれにあたります。この文書では Sink 型 DAQ コンポーネントについて解説します。

Sink 型 DAQ コンポーネントは単独で存在してもデータを受け取ることができないので作ってもそれだけでは正常に動作するかどうか確認することはできません。データを送ってくれる DAQ コンポーネントが必要です。この文書では「ソース型コンポーネント開発マニュアル」[3] にある Source 型 DAQ コンポーネント (EchoReader) からデータを受け取る DAQ コンポーネントを作成します。

2 開発方法

新しい DAQ コンポーネントを開発する方法には

- 既にある実装を改良する
- DAQ-Middleware が提供する Skeleton コンポーネントを使って開発する

のふたつのスタイルがあります。後者の Skeleton コンポーネントとはコアロジックが実装されていない DAQ コンポーネントです。ここでは Skeleton コンポーネントを使って開発する方法を説明します。

3 DAQ-Middleware での状態と状態遷移、およびエラー処理

DAQ-Middleware で定義されている状態、および状態遷移図を図 1 に示します。

DAQ コンポーネントにはその状態にある間、繰り返し呼ばれるメソッドおよび状態遷移するときに 1 回呼ばれるメソッド、の 2 種類のメソッドがあります。図 1 中 `daq_dummy()` および `daq_run()` はある状態にある間中、繰り返し呼ばれるメソッドで、`daq_configure()`、`daq_start()`、`daq_pause()`、`daq_restart()`、`daq_stop()`、`daq_unconfigure()` の各メソッドが状態遷移するときに 1 回呼ばれるメソッドです。DAQ コンポーネントを実装するときにはこれらのメソッドを実装します。

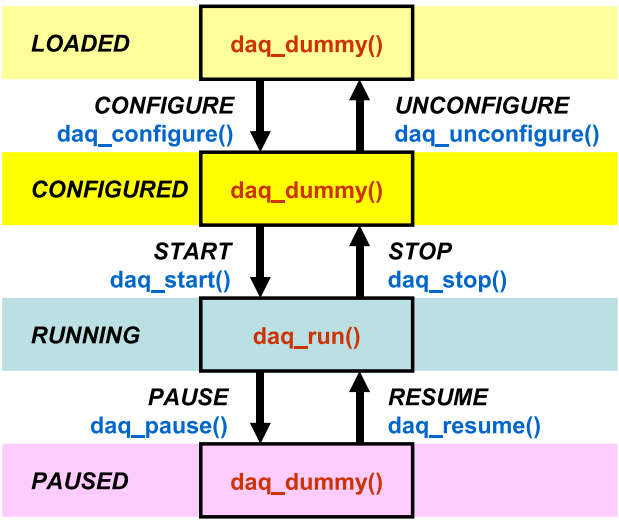

図 1　DAQ-Middleware で定義されている状態、および状態遷移。

コンポーネントがエラーを検出した場合 (整合性のないデータを受け取った、上流コンポーネントから送られてきたデータ数とヘッダーに書かれたデータ数があわないなどの場合)、コンポーネントはいきなり exit するのではなく DAQ オペレータにエラーが発生したことを報告し、自身はアイドル状態になることになっています。エラーを DAQ オペレータに報告するためのメソッドとして `fatal_error_report()` が DAQ-Middleware 内で実装されていますので DAQ コンポーネントを実装するときにはエラーの際にはこのメソッドを呼ぶようにします。

DAQ-Middleware で定義されている状態、および状態遷移、`fatal_error_report` については「Source 型コンポーネント開発マニュアル」も参照してください。

4　この文書で実装する Sink 型コンポーネントについて

では実際に Sink 型コンポーネントを書いてみましょう。ここでは

1. まず上流コンポーネントからデータを読むコンポーネント (ヒストグラム等は作成しない) を作って
2. そのコンポーネントを改良してヒストグラムを書くコンポーネントを作る

という道筋で書いてみることにします。

ここでは既に述べたように Skeleton コンポーネントを使って開発していきます。

4.1　上流データコンポーネント

Sink 型 DAQ コンポーネントは上流のコンポーネントからデータを受け取ってなんらかの仕事をするコンポーネントです。したがって、データを送ってくれるコンポーネントがないと作成

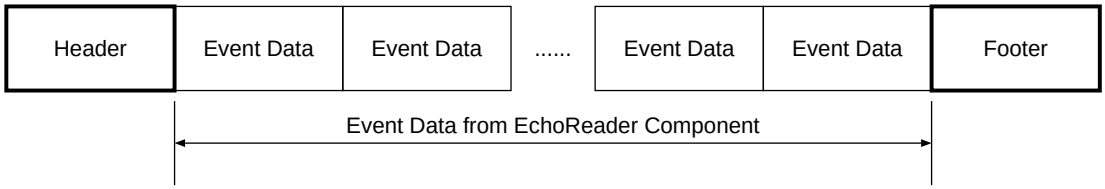

図 2　EchoReader コンポーネントから送られてくるデータフォーマット。各 Event Data は 4 バイトの整数で little endian で送られてくる。Data 数は 100 個 (400 バイト)。ヘッダー, フッターはそれぞれ 8 バイトでそのフォーマットについては図 3 を参照。

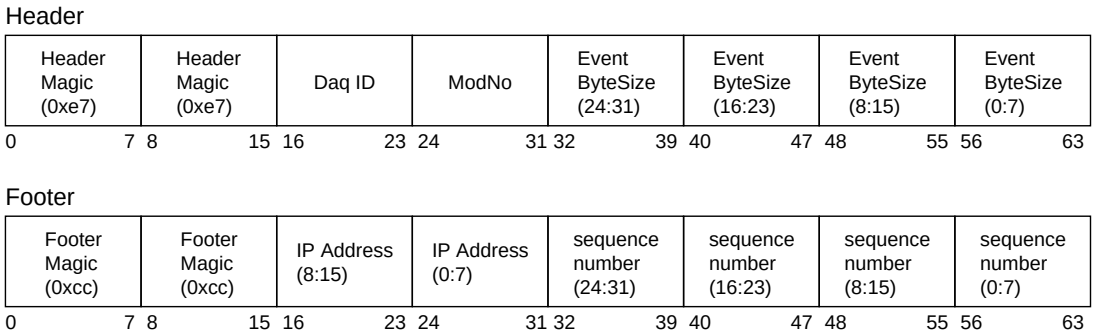

図 3　ヘッダーおよびフッターのフォーマット。

しても正常に動作するかどうか確認できません。ここでは「Source 型コンポーネント開発マニュアル」で作成した EchoReader コンポーネントを上流コンポーネントとして使用します。まず、EchoReader コンポーネントが送ってくるデータのフォーマットを確認しておきます。

4.2 データフォーマットの確認

EchoReader コンポーネントから送られてくるデータフォーマットを図 2 に示します。またこの図にあるヘッダー、フッターのフォーマットを図 3 に示します。このヘッダー、フッターは現在 J-PARC/MLF で使われているヘッダー、フッターと同一です。このうち EchoReader コンポーネントではヘッダー中の DAQ ID、ModNo は常に 0 にセットされています。またフッター中の IP Address も常に 0 にセットされています。

ヘッダー中の EventByteSize には、EchoReader コンポーネントが送ろうとしたイベントデータ長がバイトを単位として入っています。したがって受け取るコンポーネントでは、受け取ったデータ長とこの EventByteSize を比較して EchoReader コンポーネントが送ろうとしたデータ全てが受け取れたかどうかチェックすることができます。またフッターの sequence number には EchoReader コンポーネントが何回データを送ったかの回数が入っています。受け取るコンポーネント側で、何回データを受け取ったか記録しておきその値を sequence number と比較することにより読みおとしがなかったかどうかのチェックができます。

4.3　実装準備

開発環境の準備については付録 A にまとめてありますのでそちらをご覧ください。この文書で実装する DAQ コンポーネントの名前を EchoMonitor と呼ぶことにします。

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

をコピーし、

- EchoMonitor.h
- EchoMonitor.cpp
- EchoMonitorComp.cpp
- Makefile.EchoMonitor

を作成します。続けて

```
for i in *EchoMonitor*; do
sed -i.bak -e 's/skeleton/echomonitor/g' \
           -e 's/Skeleton/EchoMonitor/g' \
           -e 's/SKELETON/ECHOMONITOR/g' $i
done
```

として上のソース中の “skeleton”, “Skeleton”, “SKELETON” の文字列を大文字小文字をあわせて “echomonitor”, “EchoMonitor”, “ECHOMONITOR” に変更します。上のコマンドではオリジナルのファイルは*.bak という名前でコピーされています。このファイルは不要ですので今後必要ないと思えば削除してかまいません。この段階で

```
make -f Makefile.EchoMonitor
```

として make できるかどうか確認します。正常に make できた場合には make を実行したディレクトリ、および../bin/ディレクトリに EchoMonitorComp という実行形式ファイルができます。エラーが起きた場合は原因を追求し、解決してください。

なお、ここで準備したファイル、および以降で開発するファイルの入手については付録を参照してください。

5 上流コンポーネントからのデータを読むコンポーネントの作成

上に述べたとおり、この文書では、まず EchoReader コンポーネントからデータを受け取るコンポーネントを作成します。受け取ったデータはデコードし、標準エラー出力に出力することにします。

5.1 EchoMonitor.h の変更

EchoReader コンポーネントからのデータを受け取るために EchoMonitor.h を以下のように変更します。

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

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

private:
    int daq_dummy();
    int daq_configure();
    int daq_unconfigure();
    int daq_start();
    int daq_run();
    int daq_stop();
    int daq_pause();
    int daq_resume();
    int reset_InPort();

    int parse_params(::NVList* list);

    int m_in_status;
    int m_out_status;
    int m_sampling_rate;
    bool m_debug;

    static const int EVENT_BYTE_SIZE = 4;
```

ここでの変更内容を説明します。

- EchoMonitor コンポーネントはデータを他のコンポーネントに送ることはないのでアウトポートに関する部分をコメントアウトしています。
- 後述する `reset_InPort()` メソッドを追加するために宣言を追加しています。
- イベントデータ長 (4 バイト) を `EVENT_BYTE_SIZE` として定義しています。この変数は daq_run() の実装で使用します。

5.2　EchoMonitor.cpp の変更

この部分は少し長いのでコードの断片に分割します。コード全体を通してみたい場合は付録をお読みになってファイルを入手して参照してください。

■アウトポートの除去

```
EchoMonitor::EchoMonitor(RTC::Manager* manager)
    : DAQMW::MlfComponent(manager),
        m_InPort("echomonitor_in",   m_in_data),
        //m_OutPort("echomonitor_out", m_out_data),

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

先にも述べましたがここで作る DAQ **コンポーネントは、他の** DAQ **コンポーネントにデータを送ることはありませんのでアウトポートに関する部分をコメントアウトしています。**

続いて daq_run() **の実装に移ります。**

■上流コンポーネントからのデータの読み込み

```
int EchoMonitor::daq_run()
{
    if (m_debug) {
        std::cerr << "*** EchoMonitor::run" << std::endl;
    }

    m_in_status = m_InPort.read(m_in_data);
```

上流コンポーネントから送られてきたデータは InPort **に蓄積されています。これを取り出すには** read() **メソッドを使って** m_InPort.read(m_in_data) **とします。これで上流コンポーネントからのデータは** m_in_data.data **配列に入ります。上流コンポーネントがヘッダー、フッターを付けて送ってきた場合にはこの配列にはイベントデータだけではなくヘッダー、フッターも含まれていることに注意してください。また** 1 **回の** m_InPort.read(m_in_data) **で読めるデータは上流コンポーネントがアウトポートに書いたデータ** 1 **回分の全てのデータです。**

■データが来ていなかった場合、およびエラーの場合の処理

```
m_in_status = m_InPort.read(m_in_data);
if ((m_in_status == BUF_TIMEOUT) && check_trans_lock()) {
    set_trans_unlock();
    return 0;
}
else if (m_in_status == BUF_TIMEOUT) {
    return 0;
}
else if (m_in_status == BUF_FATAL) {
    fatal_error_report(USER_ERROR1, -1);
    return 0;
```

```
}
```

(1 行めのコードは上の「上流コンポーネントからのデータの読み取り」の最後の行と重複しています)

m_InPort.read(m_in_data) は読めるデータがなかった場合には BUF_TIMEOUT を返すのでこれを確認します。読めるデータが無かった場合には check_trans_lock() を実行し、STOP コマンドがきていたかどうか確認します (check_trans_lock() は STOP コマンドがきていたら true を返します)。STOP コマンドがきていた場合には状態遷移のロックを外し次の状態へ遷移できるようにして daq_run() を終了します。STOP コマンドが来ていなかった場合には状態遷移のロックを外さずに単に daq_run() を終了します。読み取りでエラーがあった場合には m_InPort.read() は BUF_FATAL を返します。この場合は fatal_error_report() を使って DAQ オペレータにエラーがあったことを通知します。これらの if 文が false であった場合には上流コンポーネントからのデータが読めたことになりますので、続けて読めたデータを処理します。それが以下のコードです。

■データ長の取得

```
/////////////// Get Data Length ///////////////
unsigned int block_byte_size = m_in_data.data.length();
unsigned int event_byte_size = block_byte_size - HEADER_BYTE_SIZE - FOOTER_BYTE_SIZE;
```

上流コンポーネントから送られてきたデータの長さは m_in_data.data.length() で取得できます。単位はバイトです。この値を block_byte_size に格納しています。EchoReader コンポーネントではイベントデータの他にヘッダーとフッターを追加して送ってきますのでイベントデータ長は、block_byte_size からヘッダー長およびフッター長を引いたものになります。この値を event_byte_size に格納しています。

■ヘッダー、フッターを使用したチェック

```
/////////////// Check Header and Footer ///////////////
unsigned char header[HEADER_BYTE_SIZE];
unsigned char footer[FOOTER_BYTE_SIZE];
for (unsigned int i = 0; i < HEADER_BYTE_SIZE; i++) {
    header[i] = m_in_data.data[i];
}
if (check_header(header, event_byte_size) == false) {
    std::cerr << "### ERROR: header invalid in EchoMonitor" << std::endl;
    fatal_error_report(HEADER_DATA_MISMATCH, -1);
    return 0;
}
for (unsigned int i = 0; i < FOOTER_BYTE_SIZE; i++) {
    footer[i] = m_in_data.data[block_byte_size - FOOTER_BYTE_SIZE + i];
}
if (check_footer(footer, m_loop) == false) {
    std::cerr << "### ERROR: footer invalid in EchoMonitor" << std::endl;
    fatal_error_report(FOOTER_DATA_MISMATCH, -1);
    return 0;
```

```
}
```

既にのべたように、今回、上流コンポーネントとして使用している EchoReader コンポーネントはイベントデータの他にヘッダーとフッターを追加して送ってきています。ヘッダー中には、EchoReader コンポーネントが送ってきたイベントデータの長さが含まれていますので、今作っているコンポーネントで受け取ったデータ長と整合しているかどうかを調べるためにヘッダーを m_in_data.data から取り出し check_header() メソッドで調べています。

また、EchoReader コンポーネントは何回データを送ったかをフッターにいれて送ってきていますので、今作っているコンポーネント中で何回データを受け取ったのかを m_loop で数えて、それと整合性がとれているかを check_footer() メソッドで調べています。m_loop のインクリメントの実装はのちほど行います。

check_header()、および check_footer() いずれも上記の整合性がとれていない場合には fatal_error_report() メソッドを使ってエラーが発生したことを DAQ オペレータに報告し daq_run() を終了します。

■イベントデータの取り出し、標準エラー出力へ出力

```
////////// Extract each event data and print to STDERR //////////
unsigned int *event_data;
for (unsigned int i = HEADER_BYTE_SIZE; i < block_byte_size - FOOTER_BYTE_SIZE; i += EVENT_BYTE_SIZE
    event_data = (unsigned int *) &m_in_data.data[i];
    std::cerr << "Event Data: " << *event_data << std::endl;
}
std::cerr << "*** Data Extraction End" << std::endl;
```

ここではイベントデータを取り出して、それを標準エラー出力に出力しています。また出力が終了したら各 daq_run() の区別が付くようにメッセージを表示しています。

■m_loop、m_total_event 変数のインクリメント

```
////////// One daq_run() with reading data almost done //////////
m_loop++;
m_total_event += event_byte_size / EVENT_BYTE_SIZE;
```

以上で上流コンポーネント EchoReader から送られてきたデータの処理が完了しましたので、m_loop 変数を 1 インクリメントします。この値は、上記 check_footer() で EchoReader コンポーネントがフッターに追加してきているシーケンス番号 (EchoReader コンポーネントが何回データを送ってきたか示す番号) と照合するのに使用します。また読めたイベント数を m_total_event に加えます。DAQ オペレータから定期的に、各コンポーネントにそのコンポーネントが何イベント処理したかの問い合わせがあります。各コンポーネントはその問い合わせに対して m_total_event の数値を処理したイベント数として DAQ オペレータに応答します。

■STOP コマンドがきていたかどうかの確認

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

return 0;
```

ここで STOP コマンドがやってきていたかどうか確認し、やってきていたら状態遷移のロックを外してから daq_run() を終了します。

以上で daq_run() の実装は終了です。

■EchoMonitor::reset_InPort() の実装

```
int EchoMonitor::reset_InPort()
{
    TimedOctetSeq dummy_data;

    int ret = BUF_SUCCESS;
    while (ret == BUF_SUCCESS) {
        ret = m_InPort.read(dummy_data);
    }
    std::cerr << "*** Monitor::InPort flushed\n";
    return 0;
}
```

STOP コマンドが発行され daq_stop() が実行されるときに Sink 型 DAQ コンポーネントが InPort に送られてきたデータを全部読んでいない場合があります。たとえばヒストグラム図の更新が頻繁である、ヒストグラム図の作成に時間がかかる、ヒストグラム図を ssh チャンネルを使って X 上に飛ばしているので表示に時間がかかるなどの場合です。この場合、その状態で次回 START が実行されると Sink 型コンポーネントが新しいデータを読まず、InPort にたまっていた前のデータを読むことになってしまいます。これを避けるために、ここでは daq_stop() に遷移したときに InPort にデータがまだ残っていれば InPort 内のデータを全部捨てる処理を行っています。このためのメソッドが reset_InPort() です。以下のように daq_stop() から reset_InPort() を呼ぶようにします。

```
int EchoMonitor::daq_stop()
{
    std::cerr << "*** EchoMonitor::stop" << std::endl;

    reset_InPort();

    return 0;
}
```

■その他のメソッド、その他のファイル

他のメソッドに追加する事項はありません。また EchoMonitorComp.cpp および Make-

file.EchoMonitor は 4.3 節で行った書き換え以外行う必要はありません。以上でデータを読みだすだけのコンポーネントの実装は終了です。

5.3 上流コンポーネントからの読みだしテスト

変更したソース、および Makefile.EchoMonitor があるディレクトリ (DaqComponents/src/ディレクトリ) で

```
make -f Makefile.EchoMonitor
```

と make を実行してください (単に make ではない点に注意してください)。実行後、カレントディレクトリ、および../bin/に EchoMonitorComp というファイル名の実行形式ファイルができることを確認してください。

正常に実行形式ファイルができたら config.xml ファイルを用意して EchoReader コンポーネント、EchoMonitor コンポーネント、および DAQ オペレータコンポーネントを起動します。

以下の内容のファイルを config.xml として DaqComponents ディレクトリに置きます。なお書き換えが必要です (すぐ下をご覧ください)。

```
<?xml version="1.0"?>
<configInfo>
  <daqOperator>
    <hostAddr>172.23.1.16</hostAddr>
  </daqOperator>
  <daqGroups>
    <daqGroup gid="group0">
      <components>
        <component cid="EchoReader0">
          <hostAddr>172.23.1.16</hostAddr>
          <hostPort>50000</hostPort>
          <instName>EchoReader0.rtc</instName>
          <execPath>/home/daq/DaqComponents/bin/EchoReaderComp</execPath>
          <confFile>/home/daq/DaqComponents/rtc.conf</confFile>
          <startOrd>2</startOrd>
          <inPorts>
          </inPorts>
          <outPorts>
            <outPort>echoReader_out</outPort>
          </outPorts>
          <params>
          </params>
        </component>
        <component cid="EchoMonitor0">
          <hostAddr>172.23.1.16</hostAddr>
          <hostPort>50000</hostPort>
          <instName>EchoMonitor0.rtc</instName>
          <execPath>/home/daq/DaqComponents/bin/EchoMonitorComp</execPath>
          <confFile>/home/daq/DaqComponents/rtc.conf</confFile>
          <startOrd>1</startOrd>
          <inPorts>
            <inPort from="EchoReader0:echoReader_out">echomonitor_in</inPort>
          </inPorts>
```

```
          <outPorts>
          </outPorts>
          <params>
          </params>
        </component>
      </components>
    </daqGroup>
  </daqGroups>
</configInfo>
```

上の config.xml で書き換えが必要な部分を順に述べます。

1. hostAddr に書いてある IP アドレスをお使いの計算機の IP アドレスに修正します。hostAddr は 3 箇所 存在します。
2. コンポーネントの絶対パスを指定している execPath の部分を実際の絶対パスに修正します。execPath は 2 箇所存在しています。
3. confFile のディレクトリ部分をお使いのディレクトリに変更します。confFile は 2 箇所存在しています。なおここで指定した rtc.conf は下で述べる run-local.py が自動生成しますので今の時点で存在している必要はありません。

config.xml の書き換えに失敗するとコンポーネントの起動に失敗しますので注意が必要です。

コンポーネントを起動するまえにカレントディレクトリに omninames-で始まるファイルがあればこれらを消しておきます。

```
rm omninames-*
```

omninames-で始まるファイルはネームサーバー omniNames が作成するファイルで、コンポーネントの登録内容等をログファイルとして保存しているものです。この中には omniNames が起動された計算機の IP アドレスなどの動作している計算機固有の情報が含まれています。したがって他の計算機上のファイルを持ってきた場合、あるいは IP アドレスを DHCP で取得している場合などで動作計算機環境が変化したときには omniNames が正常に起動しないことがあります。それを解消するためにこのファイルを消しておきます。

コンポーネントの起動は run-local.py コマンドを使って以下のように行います。

```
daq% cd /home/daq/DaqComponents
daq% ls config.xml
# (config.xml が存在するかどうか確認)
daq% ./run-local.py -c
# (起動コマンド)
```

-c オプションを指定すると端末エミュレータからキーボードの数字キーを使って DAQ オペレータから各コンポーネントに START、STOP 等のコマンドを送るモード (コンソールモード) で起動が行われます。この他に Web から mod_python を通じて DAQ オペレータ経由でコマンドを送るモードも存在しますが、ここでは使用しません。run-local.py を使用した場合には各コ

ンポーネントからの標準エラー出力は/tmp/以下に log. コンポーネント実行ファイル名 というファイル名形式のファイルに保存されます。たとえば EchoMonitorComp からの標準エラー出力は/tmp/log.EchoMonitorComp に記録されます。このファイルはコンポーネントが再起動されるごとに上書きされます。

エラーメッセージ

```
./bin/DaqOperatorComp: error while loading shared libraries: /home/daq/lib/libSock.so:
 cannot restore segment prot after reloc: Permission denied
```

が出た場合の対処方法については付録 A の SELinux の節をご覧ください。

正常にコンポーネントが起動すると下の図のようになります。

```
 Current State: LOADED
 Command:  0:configure  1:start  2:stop  3:unconfigure  4:pause  5:resume
0 0
```

LOADED は各コンポーネントが LOADED 状態にあることを示します。Command:の行が使用可能なコマンドを示しています。最後の 0 0 の行は、DAQ オペレータが今起動しているふたつの DAQ コンポーネントから取得したそれぞれの DAQ コンポーネントで扱ったイベント数を示しています。この状態で "0" を押すと configure が実行され "Current Status" が CONFIGURED になります。configure を実行後 "1" を押すと run number を聞いてきますので適当な数字を入力します。run number の入力が終わると自動的に RUNNING 状態になり各コンポーネントが動作しはじめます。2 を押すと動作が終了して "Current Status" が CONFIGURED になります。今の段階の EchoMonitor コンポーネントはどんどん/tmp/log.EchoMonitor にイベントデータを出力しますので、あまり長い時間稼働させておくとディスクスペースが無駄になりますので注意してください。2 を押して STOP 状態になったあと終了するには Control-C を押します。こうすると

```
omniORB: ERROR -- the application attempted to invoke an operation
 on a nil reference.
No service connected.
```

というメッセージが出力されます。この状態で各コンポーネントプロセスは終了していますが、run-local.py 中で起動された omniNames プロセスはまだ残っています。リモート計算機上で作業している場合 Linux ではこのような状態の場合 logout しようとしても制御がローカルにもどってきません。logout する前に

```
pkill omniNames
```

として omniNames プロセスを kill してから logout してください。

/tmp/log.EchoMonitor には daq_run() で実装したイベントデータを標準エラー出力に書き出したデータがありますので確認してください。たとえば下のようになっています。

```
*** EchoMonitor::configure
param list length:0
*** EchoMonitor::start
Event Data: 69
Event Data: 60
（途中略）
Event Data: 64
Event Data: 55
*** Data Extraction End
Event Data: 80
Event Data: 66
（途中略）
Event Data: 52
Event Data: 62
*** Data Extraction End
```

EchoReader は平均 60、標準偏差 10 の整数値 Gauss 分布のデータを、1 回につき 100 個送ってきているはずですので “Event Data: ” にかかれているデータが正常そうかどうか、また 1 回の daq_run() で 100 個のデータが読めていたかどうか確認してみてください。

以上で EchoReader コンポーネントからデータを受け取ることができる EchoMonitor コンポーネントができました。Sink 型コンポーネントの最重要課題の一つである上流コンポーネントからのデータの受け取りはできたことになります。では続けて受け取ったデータを使って ROOT を使ってヒストグラムを書いてみることにしましょう。

6 ROOT を使ってヒストグラム図を画面に表示する

これまでの実装で、EchoReader コンポーネントからデータを受け取ることができる EchoMonitor コンポーネントができました。ヒストグラムを書くには受け取ったデータをもとにヒストグラムを書けばよいことになります。ここではヒストグラムを書くのに ROOT を使うことにします。

参考文献「ROOT を用いたモニターコンポーネント開発」 [4] もあわせてご覧ください。この例題ではこの参考文献に沿って実装することにして、daq_start() するたびにヒストグラム図を新規に書きはじめることにします。

6.1 Makefile.EchoMonitor の変更

まず `root-config` コマンドが使えるように

- `ROOTSYS` 環境変数に root が存在する基準ディレクトリ (`/usr/local/root` など) を設定する
- `root-config` コマンドが使えるようにシェルの `PATH` 変数に`$ROOTSYS/bin` を追加する

としておきます。いつも設定しておくようにするためには、シェルとして bash を使っているならたとえば `$HOME/.bash_profile` に次のように書きます。

```
ROOTSYS=/usr/local/root
export ROOTSYS
PATH=$PATH:$HOME/bin:$ROOTSYS/bin

if [ -z "$LD_LIBRARY_PATH" ]; then
    LD_LIBRARY_PATH=$ROOTSYS/lib
    export LD_LIBRARY_PATH
else
    LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ROOTSYS/lib
    export LD_LIBRARY_PATH
fi
```

ここで書いた設定はすぐには有効にならないので、すぐ有効にするには一度 logout するか source .bash_profile コマンドを実行します。

続いて Makefile.EchoMonitor を変更して ROOT のインクルードファイル、およびライブラリファイルを見ることができるようにします。ここでは

- root-config --glibs を使って ROOTLIBS 変数にリンクするライブラリを指定する
- root-config --incdir を使って ROOTINC 変数にインクルードファイルがある場所を指定する

という方針で変更する場合の例を示します。以下の変更を Makefile.EchoMonitor に追加します。

```
ROOTLIBS = ‘$(ROOTSYS)/bin/root-config --glibs‘
ROOTINC  = -I ‘$(ROOTSYS)/bin/root-config --incdir‘

CXXFLAGS = ‘rtm-config --cflags‘ $(ROOTINC)
```

および

```
EchoMonitorComp: EchoMonitorComp.o $(OBJS)
    $(CXX) -o $@ $(OBJS) EchoMonitorComp.o $(LDFLAGS) $(ROOTLIBS)
    cp $@ $(BINDIR)/$@
```

6.2 EchoMonitor.h の変更

EchoMonitor.h で次のインクルードファイルを読むように変更します。これは ROOT のインクルードファイルです。

```
////////// ROOT Include files //////////
#include "TH1.h"
#include "TCanvas.h"
#include "TApplication.h"
```

また以下の変数を宣言します。m_bin はビン数、m_min はヒストグラムの最小値、m_max はヒストグラムの最大値を指定するのに使用します。m_monitor_update_rate は 1 回の daq_run() ごと

にヒストグラムを書くのではなくこの回数毎にヒストグラムをアップデートするのに使用します。

```
TCanvas *m_canvas;
TH1F    *m_histo;
int      m_bin;
double   m_min;
double   m_max;
int      m_monitor_update_rate;
```

6.3 EchoMonitor.cpp の変更

EchoMonitor.h **で宣言した変数の初期値を与えるため** EchoMonitor::EchoMonitor **クラスのコンストラクタを以下のように変更します。**

```
Monitor::EchoMonitor(RTC::Manager* manager)
: DAQMW::MlfComponent(manager),
    m_InPort("echomonitor_in",   m_in_data),
    //m_OutPort("echomonitor_out", m_out_data),

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

    m_debug(false),
    m_canvas(0),
    m_histo(0),
    m_bin(60),
    m_min(0.0),
    m_max(120.0),
    m_monitor_update_rate(10)
```

ヒストグラム図を daq_start() **するごとに新規に作るために以下の行を** daq_start() **に追加します。**

```
if (m_canvas) {
    delete m_canvas;
    m_canvas = 0;
}
m_canvas = new TCanvas("c1", "canvas", 10, 10, 300, 320);

if (m_histo) {
    delete m_histo;
    m_histo = 0;
}
m_histo = new TH1F("histo", "EchoMonitor", m_bin, m_min, m_max);
```

また daq_run() **でイベントデータを取り出して標準エラー出力に出力していた部分を以下のように変更します。**

```
////////// Extract each event data and fill to histogram data //////////
unsigned int *event_data;
for (unsigned int i = HEADER_BYTE_SIZE; i < block_byte_size - FOOTER_BYTE_SIZE; i += EVENT_BYTE_SIZE
    event_data = (unsigned int *) &m_in_data.data[i];
    m_histo->Fill(*event_data);
}
```

```
if (m_loop % m_monitor_update_rate == 0) {
    m_histo->Draw();
    m_canvas->Update();
}
```

変更点は

- **イベントデータを取り出したらそれを標準エラー出力に出すのではなく** `Fill()` **を使ってヒストグラムにフィルする**
- **イベントデータが読めた** daq_run() **が** m_monitor_update_rate **回実行されるごとにヒストグラム図を更新する**

です。2 番目の変更は計算機負荷を考慮したものです。もっと頻繁に更新したい場合は `m_monitor_date_rate` **の値を小さい値に変更します。**[*1]

6.4 EchoMonitorComp.cpp の変更

EchoMonitorComp.cpp **にある** `main()` **関数に以下のように** `TApplication()` **を追加します。**

```
int main (int argc, char** argv)
{
    RTC::Manager* manager;
    manager = RTC::Manager::init(argc, argv);

    TApplication theApp("App", &argc, argv);

    // Initialize manager
    manager->init(argc, argv);
```

`TApplication()` **の詳細については** ROOT **のマニュアルをご覧ください。**

6.5 make の実行

以上でヒストグラムを画面に表示できるようにする変更は終わりました。

```
make -f Makefile.EchoMonitor
```

として make **してください。正常に** `EchoMonitorComp` **という実行ファイルができたら** `ldd EchoMonitorComp` **コマンドで動作に必要な全てのシェアードライブラリが読めるかどうか確認してください。特に** ROOT **を**/usr/local/root **以下にインストールしている場合は** `LD_LIBRARY_PATH` **に**`/usr/local/root/lib` **が入っている必要があります。**

[*1] **ソースコードの改変なしにヒストグラム更新頻度を変更したい場合はたとえば** Condition **データベースを使用します。**Condition **データベースについては「**Condition **データベースの開発マニュアル」**[2] **をご覧ください。またこの文書の** 7 **節で、ヒストグラムのパラメーターを** Condition **データベースを使って与えるように改良しますのでそちらも参照してください。**

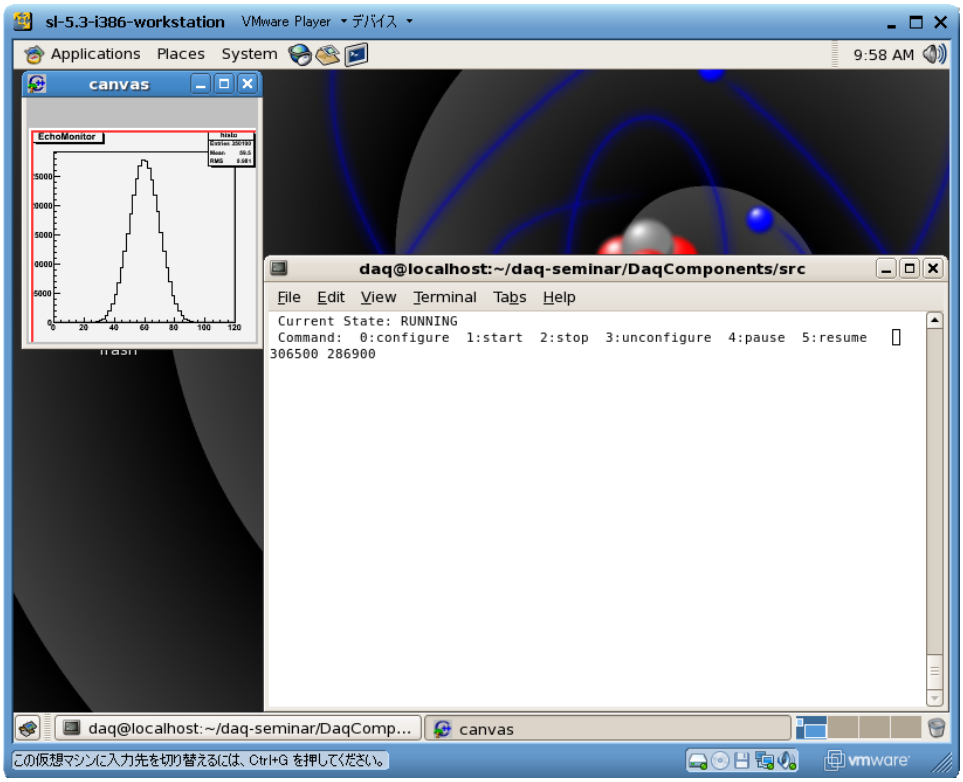

図 4　vmplayer を使って EchoReader からデータを読み取りヒストグラムを書かせているところ。ヒストグラムの作成には ROOT を使用している。

6.6　ヒストグラム表示テスト

ヒストグラム表示テストは修正前のデータ読み取りテストの場合と同様に config.xml を用意して `run-local.py -c` コマンドで行います。config.xml はデータ読み取りテストで行ったときと同じものを使います。

図 4 はヒストグラム表示テストを vmplayer 上で行っているところのスクリーンショットです。

7　パラメータの Condition データベース化

7.1　実装

いままでのべたヒストグラムでは以下の項目がソースコード内で決め打ちになっていました。

- **ヒストグラムのビン数**
- **ヒストグラムの最小値**
- **ヒストグラムの最大値**
- **ヒストグラムを更新する頻度**

この節では Condition データベースを使って、ソースコードの改変をしなくてもこれらのパラメータを与えられるように変更します。

Condition データベースについてはドキュメント「Condition データベースの開発マニュアル」[2] をご覧ください。ここではこのマニュアル中の「class を用いた実装」に沿って実装を行います。パラメーターをセットするタイミングは daq_start() 時に行うことにします。

新規に ConditionEchoMonitor.h と ConditionEchoMonitor.cpp ファイルを作成しこの中でパラメーターを保持する変数およびパラメータを取得するクラス ConditionEchoMonitor クラスを作ります。また、DAQ コンポーネントが Condition データベースを読めるようにするためには JsonSpirit ライブラリをリンクする必要があります。そこで増えたソースファイルの分もあわせて Makefile.EchoMonitor を変更します。変更点は

- JsonSpirit ライブラリをリンクするようにする
- ConditionEchoMonitor.o もリンクする必要があるのでこれを OBJS に追加する

の 2 点です。

```
JSON_DIR  = ../../json_spirit_v2.06/json_spirit
JSON_INC  = -I$(JSON_DIR)
JSON_LIB  = -lJsonSpirit

CXXFLAGS = `rtm-config --cflags` $(ROOTINC) $(JSON_INC)
LDFLAGS  = `rtm-config --libs` -L$(JSON_DIR) $(JSON_LIB)
```

```
OBJS     = EchoMonitor.o $(SKEL_OBJ) $(IMPL_OBJ)
OBJS     += ConditionEchoMonitor.o
```

パラメーターのデータ構造は、ConditionEchoMonitor.h 内で構造体 monitorParam で定義することにします。

```
struct monitorParam {
    unsigned int histogram_n_bin;
    unsigned int histogram_min;
    unsigned int histogram_max;
    unsigned int monitor_update_rate;
};
```

次に condition.json ファイルを読み monitorParam 構造体変数にセットするメソッドを ConditionEchoMonitor.cpp に追加します。

```
ConditionEchoMonitor::getParam(std::string prefix, monitorParam* monitorParam)
{
    setPrefix(prefix);
    unsigned int histogram_n_bin;
    unsigned int histogram_min;
    unsigned int histogram_max;
    unsigned int monitor_update_rate;

    if (find("histogram_n_bin", &histogram_n_bin)) {
```

```
        monitorParam->histogram_n_bin = histogram_n_bin;
    }
    else {
        std::cerr << prefix + " histogram_n_bin not fould" << std::endl;
        return false;
    }

    if (find("histogram_min", &histogram_min)) {
        monitorParam->histogram_min = histogram_min;
    }
    else {
        std::cerr << prefix + " histogram_min not fould" << std::endl;
        return false;
    }

    if (find("histogram_max", &histogram_max)) {
        monitorParam->histogram_max = histogram_max;
    }
    else {
        std::cerr << prefix + " histogram_max not fould" << std::endl;
        return false;
    }


    if (find("monitor_update_rate", &monitor_update_rate)) {
        monitorParam->monitor_update_rate = monitor_update_rate;
    }
    else {
        std::cerr << prefix + " monitor_update_rate not fould" << std::endl;
        return false;
    }


    return true;
}
```

以上で ConditionEchoMonitor **クラスの準備ができました。**

続けて EchoMonitor **側で** Condition **データベースを使ってヒストグラムのパラメーターを取得するようにします。まず、**Condition **データベースのファイル名** CONDITION_FILE**、およびパラメーターを保持する構造体** m_monitorParam **を** EchoMonitor.h **に追加**:

```
////////// Condition database //////////
static const std::string CONDITION_FILE;
monitorParam m_monitorParam;
```

さらに EchoMonitor.cpp **で** CONDITION_FILE **に値を代入します。**

```
const std::string EchoMonitor::CONDITION_FILE = "./condition.json";
```

つぎにこのクラスを使用するように DAQ **コンポーネントのソースを変更します。まず、**EchoMonitor.cpp **で** set_condition() **関数を追加し、**ConditionEchoMonitor **クラスを使ってパラメータを取得します。**

```
int set_condition(std::string condition_file, monitorParam *monitorParam)
{
    ConditionEchoMonitor conditionEchoMonitor;
    conditionEchoMonitor.initialize(condition_file);
    if (conditionEchoMonitor.getParam("common_EchoMonitor_", monitorParam)) {
        std::cerr << "condition OK" << std::endl;
    }
    else {
        throw "EchoMonitor condition error";
    }

    return 0;
}
```

daq_start() で set_condition() を呼ぶようにします。

```
try {
    set_condition(CONDITION_FILE, &m_monitorParam);
}
catch (std::string error_message) {
    std::cerr << error_message << std::endl;
    fatal_error_report(USER_ERROR1, -1);
    return 0;
}
catch (...) {
    std::cerr << "unknown error" << std::endl;
    fatal_error_report(USER_ERROR1, -1);
    return 0;
}
```

さらにヒストグラムのビンの数、最小値、最大値としてこの取得した値を使うように、TH1F() の引数を変更します。またヒストグラムを更新するタイミングを決めているところも変更します。

```
m_histo = new TH1F("histo", "EchoMonitor",
            m_monitorParam.histogram_n_bin,
            m_monitorParam.histogram_min,
            m_monitorParam.histogram_max);
```

```
if (m_loop % m_monitorParam.monitor_update_rate == 0) {
    m_histo->Draw();
    m_canvas->Update();
}
```

7.2 Condition データベースを使ったヒストグラムのテスト

パラメーターの値を condition.xml で与えます。condition.xml は/home/daq/DaqComponents/ディレクトリに置きます。

```
<?xml version="1.0" encoding="utf-8" ?>
<condition>
    <common>
```

```
        <EchoMonitor>
            <histogram_n_bin>100</histgogram_n_bin>
            <histogram_min>0</histogram_min>
            <histogram_max>200</histogram_max>
            <monitor_update_rate>10</monitor_update_rate>
        </EchoMonitor>
    </common>
</condition>
```

コンポーネントはこの xml **ファイルを読むのではなく、**JSON **形式に変換したファイル** condition.json **を読みます。**JSON **形式への変換は** condition_xml2json **コマンドを使って行います。**

```
% cd /home/daq/DaqComponents
  condition.xml をテキストエディターで作る
% ./condition_xml2json condition.xml
```

このコマンドで condition.json **ファイルができます。このコマンドはシェルスクリプトでその内部で** Xalan **コマンドを使っていますので** xalan **パッケージが必要です。**

起動は、Condition **データベース化前と同様に** run-local.py -c **コマンドで行います。また** config.xml **ファイルに変更するところはありません。ヒストグラムビン数、最小値、最大値等が上の** condition.xml **の値になっていることを確認してください。**

参考文献

[1] DAQ-Middleware Home page http://greentea.kek.jp/daqm/

[2] **安芳次、**Condition **データベースの開発マニュアル、**2009 **年** 7 **月** 3 **日、** http://greentea.kek.jp/daqm/docs/ConditionDevManual.pdf

[3] **仲吉一男、**Source **型コンポーネント開発マニュアル、**2009 **年** 7 **月、** http://greentea.kek.jp/daqm/docs/source-comp.pdf

[4] **仲吉一男、**ROOT **を用いたモニターコンポーネント開発、**2009 **年** 6 **月、** http://greentea.kek.jp/daqm/docs/monitor-root.pdf

付録 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-daqmiddleware.repo 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
```

付録 B　ソースコードの入手法

この文書で開発した Sink **型コンポーネントは** http://greentea.kek.jp/daqm/src/sink-comp.tar.gz **にまとめてありますのでご利用ください。**

このソースコードだけでは実際に動作するコンポーネントは作成できません。付録 A **にある準備をして使用してください。**

この tarball **を展開すると** echo-monitor **ディレクトリが作成されその中に以下のディレクトリ、ファイルが作成されます。**

```
echo-monitor
|-- condition.xml
|-- config.xml
|-- pre
|   |-- EchoMonitor.cpp
|   |-- EchoMonitor.h
|   |-- EchoMonitorComp.cpp
|   `-- Makefile.EchoMonitor
|-- data
|   |-- EchoMonitor.cpp
|   |-- EchoMonitor.h
|   |-- EchoMonitorComp.cpp
|   `-- Makefile.EchoMonitor
|-- histogram
|   |-- EchoMonitor.cpp
|   |-- EchoMonitor.h
|   |-- EchoMonitorComp.cpp
|   `-- Makefile.EchoMonitor
`-- condition
    |-- ConditionEchoMonitor.cpp
    |-- ConditionEchoMonitor.h
    |-- EchoMonitor.cpp
    |-- EchoMonitor.h
    |-- EchoMonitorComp.cpp
    `-- Makefile.EchoMonitor
```

各ディレクトリには以下のものが入っています。

- pre **ディレクトリには** 4.3 **節の** sed **コマンドで** skeketon **等の文字列を** echomonitor **等に置き換えたソースが入っています。**
- data **ディレクトリには** 5 **節で開発した上流** DAQ **コンポーネントからデータを読み標準エラー出力に出力するコンポーネントのソースが入っています。**
- histogram **ディレクトリには** 6 **節で開発した** ROOT **でヒストグラムを書くコンポーネントのソースが入っています。**
- condition **ディレクトリには** 7 **節で開発した** Condition **データベースを使用してヒストグラムのパラメーターを与えるコンポーネントのソースが入っています。**
- config.xml **はこの文書で使用した** config.xml **ファイルです。本文中でも述べましたが**

この config.xml を使用する場合は必ず変更しなければならない行が複数ありますのでご注意ください。変更箇所については 5 節をご覧ください。

- condition.xml はこの文書の 7 節で使用した condition.xml です。condition.xml は変更後、./condition_xml2json condition.xml とコマンドを実行して condition.json ファイルに変換する必要がある点にご注意ください。

これらのソースを/home/daq/DaqComponents/src/ディレクトリにコピーするかシンボリックリンクを作成するなどしてご利用ください。