DAQ-Middleware講習会
DAQコンポーネント開発

千代浩司
高エネルギー加速器研究機構
素粒子原子核研究所

もくじ

- ドキュメンテーション
- 開発環境セットアップ
- DAQコンポーネント作成方法
  - DAQコンポーネント状態遷移
  - コンポーネント間データフォーマット
  - エラー時の処理
  - InPort, OutPortの読み書き
  - 開発環境の使い方（Makefile等）
  - デモ

ドキュメンテーション

- DAQ-Middleware 1.1.0 技術解説書

  http://daqmw.kek.jp/docs/DAQ-Middleware-1.1.0-Tech.pdf

- DAQ-Middleware 1.1.0開発マニュアル

  http://daqmw.kek.jp/docs/DAQ-Middleware-1.1.0-DevManual.pdf

DAQ-Middlewareセットアップ

- Scientific Linux (CentOS, RHEL) 5.xの場合は

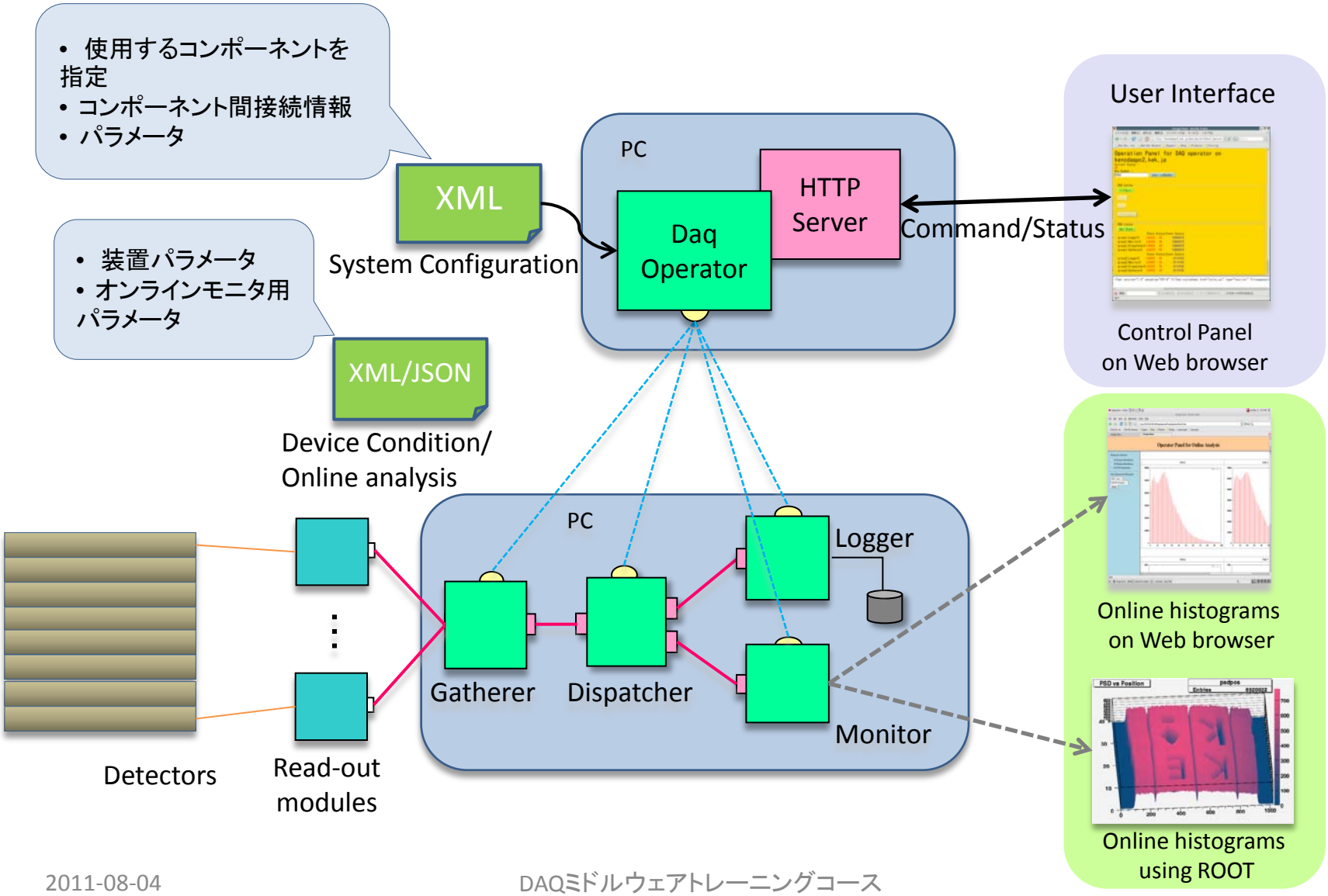
  % wget http://daqmw.kek.jp/src/daqmw-rpm

  % su

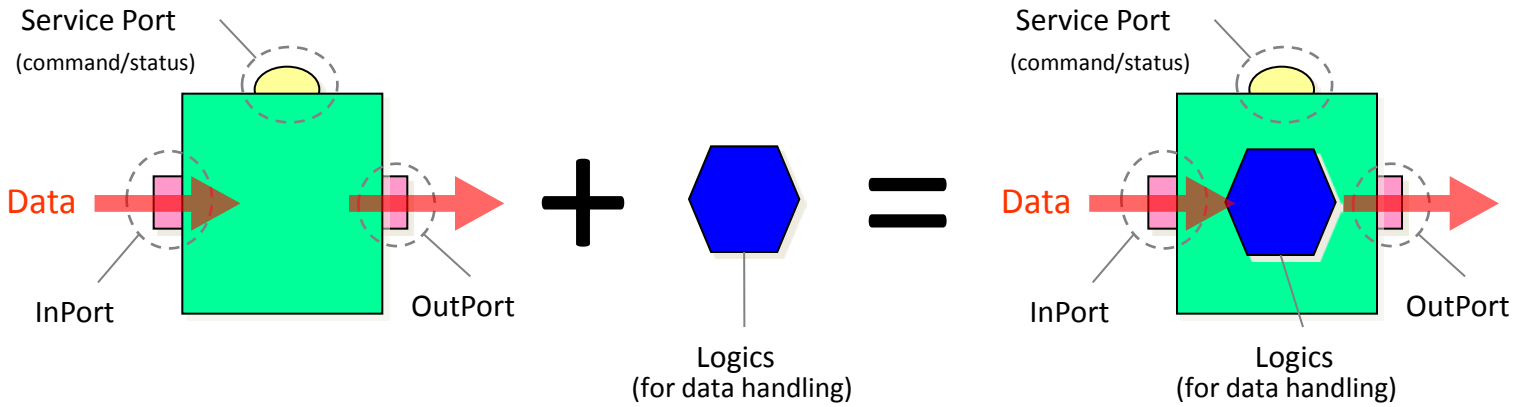
  root# sh daqmw-rpm install

  ファイル一覧はrpm -ql DAQ-Middleware
- 今回使用するVMwareイメージは上のコマンドで作りました。
- その他のOSの場合は、依存物をセットしたうえでソースからインストールすることになります。

基本DAQモデル


• 使用するコンポーネントを指定
• コンポーネント間接続情報
• パラメータ
XML
System Configuration
• 装置パラメータ
• オンラインモニタ用パラメータ
XML/JSON
Device Condition/
Online analysis
PC
Daq Operator
HTTP Server
Command/Status
User Interface
Control Panel
on Web browser
Online histograms
on Web browser
Online histograms
using ROOT
PC
Logger
Gatherer
Dispatcher
Monitor
Detectors
Read-out
modules

DAQコンポーネント

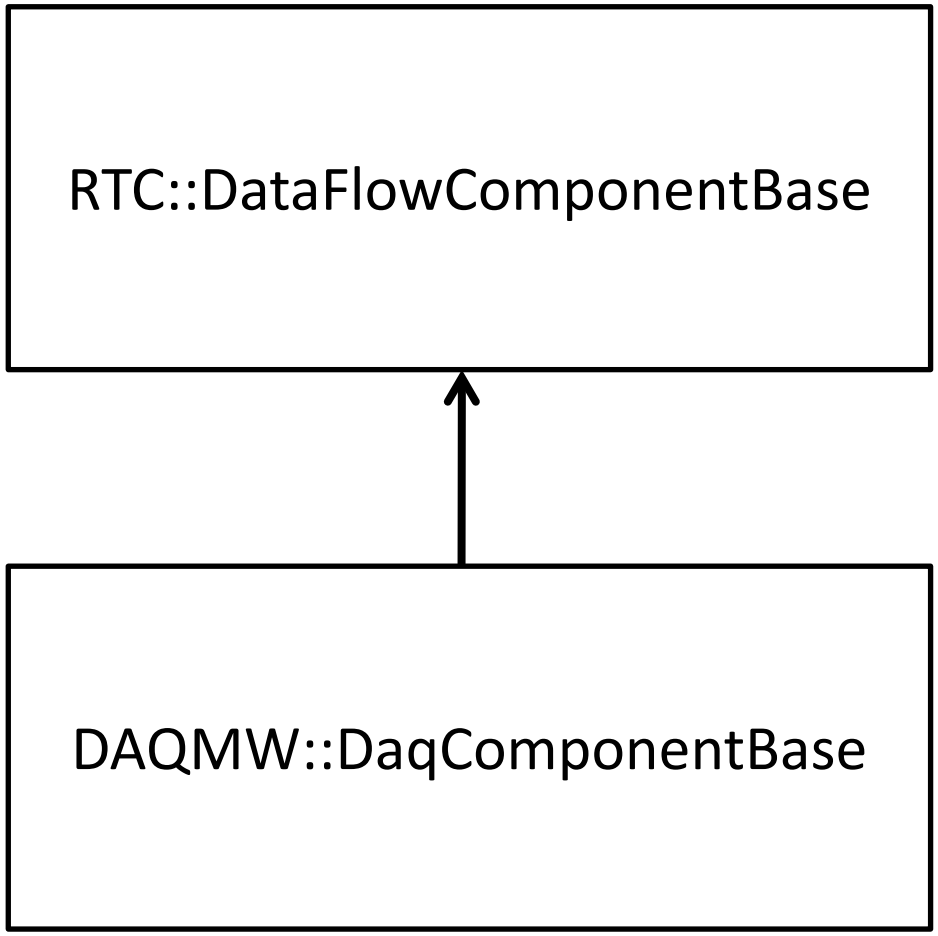


- DAQコンポーネントを組み合わせてDAQシステムを構築する
- データ転送機能、ランコントロール、システムコンフィギュレーション機能はDAQ-Middlewareで実装済み。
  - データを下流に送るにはOutPortに書く。
  - 上流からのデータを読むにはInPortを読む。
- ユーザーはコアロジックを実装することで新しいコンポーネントを作成できる。
  コアロジックの例：
  - リードアウトモジュールからのデータの読み取りロジック
  - ヒストグラムの作成ロジック

DAQ-Middlewareを使った DAQシステム開発のながれ

- コンポーネント作成
- configuration fileの作成
- コンポーネント起動、DaqOperator起動
- DaqOperatorに対して指示をだす

クラス

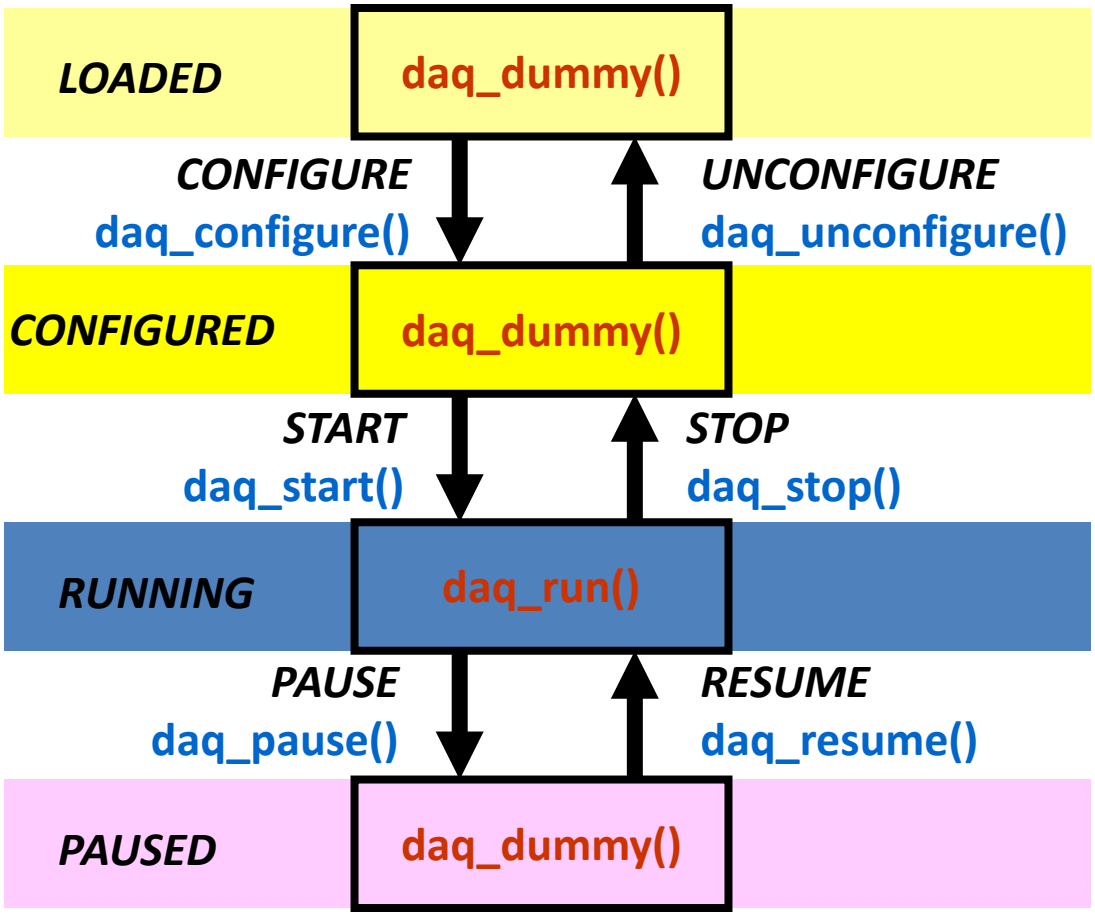


1コンポーネントに必要なソースファイル

Skeletonという名前のコンポーネントの場合

- Skeleton.h (DaqComponentBaseを継承。Skeletonクラス）
- Skeleton.cpp (各状態ロジックを実装)
- SkeletonComp.cpp (main()がここにある)
- Makefile
- その他分離したくなったファイル

コンポーネント状態遷移

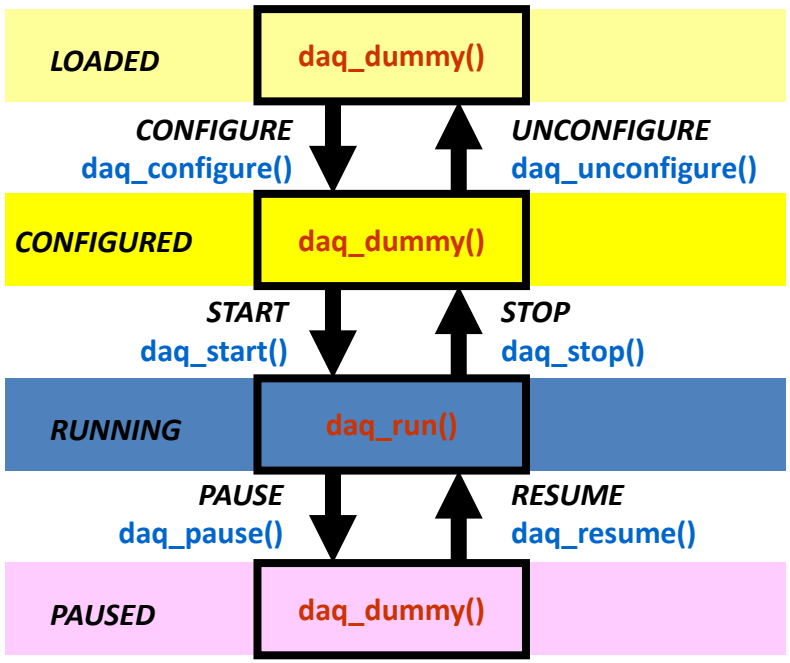


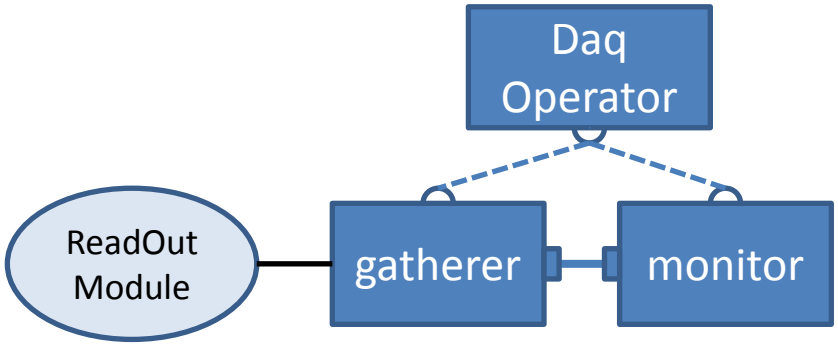
各状態にあるあいだその
関数がくりかえし呼ばれる。

状態遷移するときは状態遷移
関数が呼ばれる。

状態遷移できるようにするためには、daq_run()等は永遠にそのなかでブロックしてはだめ。
（例：Gathererのソケットプログラムでtimeoutつきにする必要がある）

コンポーネント状態遷移





Gatherer

daq_start(): リードアウトモジュールに接続
daq_run(): リードアウトモジュールからデータを読んで後段コンポーネントにデータを送る
daq_stop(): リードアウトモジュールから切断。

Monitor

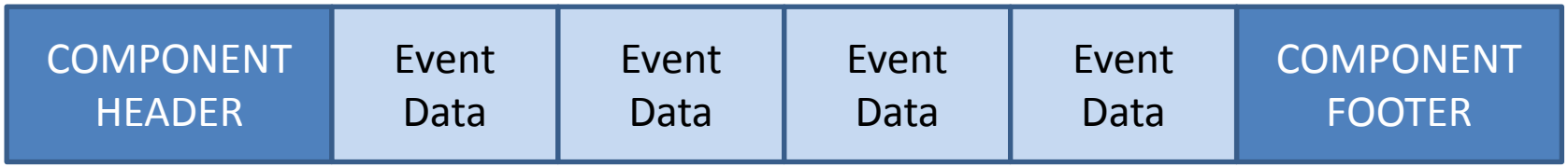
daq_start(): ヒストグラムデータの作成
daq_run(): 上流コンポーネントからデータをうけとり、デコードしてヒストグラムデータをアップデートする。定期的にヒストグラム図を書く
daq_stop(): 最終データを使ってヒストグラム図を書く

コンポーネント実装方法

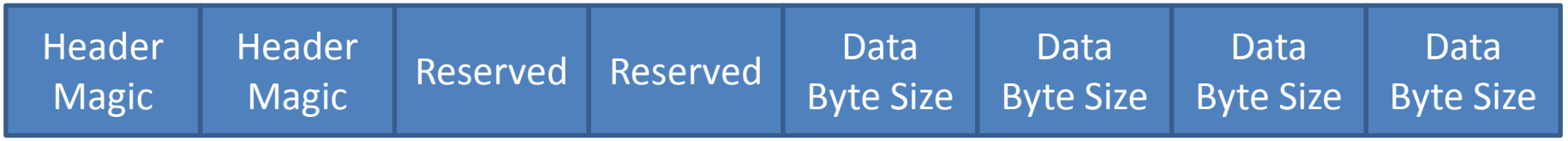
各メソッドを実装することでコンポーネントを作成する

- daq_configure()
- daq_start()
- daq_run()
- daq_stop()
- daq_unconfigure()

コンポーネント間のデータフォーマット



Component Header



Data Byte Sizeには下流コンポーネントに何バイトのデータを送ろうとしたかを入れる
下流側ではDataByteSizeを読んでデータが全部読めたかどうか判断する

Component Footer

Sequence Numberにデータを送るのは何回目かを入れる
下流側では受け取った回数を自分で数えておいて、Sequence Numberとあうかどうか確認する

コンポーネント間のデータフォーマット

Header Magic	Header Magic	Reserved	Reserved	Data Byte Size	Data Byte Size	Data Byte Size	Data Byte Size

Footer Magic	Footer Magic	Reserved	Reserved	Seq. Num	Seq. Num	Seq. Num	Seq. Num

Reservedのバイトはユーザが使用してもよい

コンポーネント間データフォーマット関連メソッド

- inc_sequence_num()
- reset_sequence_num()
- get_sequence_num()

- set_header(unsigned char *header, unsigned int data_byte_size)
- set_footer(unsinged char *footer) /* API Changed */

- check_header(unsigned char *header, unsigned received_byte)
- check_footer(unsigned char *footer) /* API Changed */
- check_header_footer(const RTC::TimedOctetSeq& in_data, unsigned int block_byte_size)

Header Magic	Header Magic	Reserved	Reserved	Data Byte Size	Data Byte Size	Data Byte Size	Data Byte Size
Footer Magic	Footer Magic	Reserved	Reserved	Seq. Num	Seq. Num	Seq. Num	Seq. Num

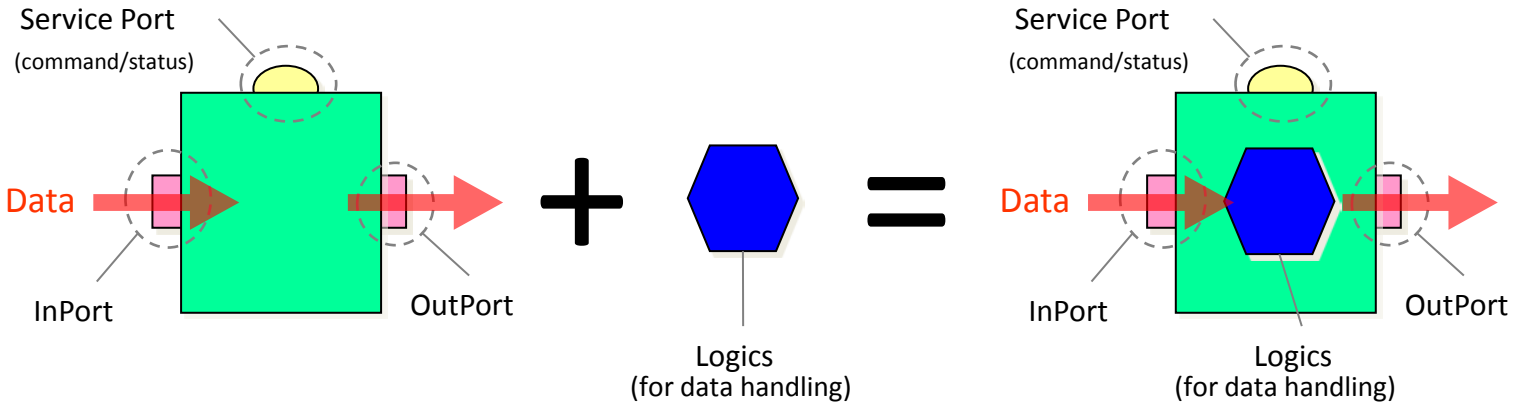
Fatal Error

- 致命的エラーが起こったらfatal_error_report()を使ってDaqOperatorへ通知する。
- DAQ-Middlewareで定義しているものとユーザーが定義できるものがある。

```
fatal_error_report(USER_DEFINED_ERROR1,
        "cannot connect to readout module");
```

- DaqOpertorに通知されたあとの動作は上位のフレームワークあるいは人が対処する(ランを停止する、再スタートするなど)

InPort, OutPort操作



Skeleton.h:

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

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

Skeleton.cpp

```
// Ctor
Skeleton::Skeleton(RTC::Manager* manager)
   : DAQMW::DaqComponentBase(manager),
     m_InPort("skeleton_in",   m_in_data),
     m_OutPort("skeleton_out", m_out_data),
```

InPort操作

bool rv = m_InPort.read()

- 読んだデータはm_in_data.data配列にデータが入る
- length = m_in_data.data.length() で長さ
  （Component Header, Footerを含めた長さ）
- 戻り値： true, false
- falseの場合は check_inPort_status(m_InPort)でInPortの状態を確認する。

  check_inPort_statusの戻り値
  - BUF_TIMEOUT: 通常リトライするようにコードを書く
  - BUF_FATAL: 通常fatal_error_report()でエラーを報告

OutPort操作

bool rv = m_OutPort.write()

- m_out_data.data.length(length)でデータ長を指定
  （Component Header, Footerを含めた長さ）
- 送るデータはm_out_data.data配列に入れる
  （Component Header, Footerを含める）
- m_OutPort.write()でデータが送られる。
- 戻り値： true, false
- falseの場合は check_outPort_status(m_OutPort)で OutPortの状態を確認する。

  check_inPort_statusの戻り値
  - BUF_TIMEOUT: 通常リトライするようにコードを書く
  - BUF_FATAL: 通常fatal_error_report()でエラーを報告

DaqOperator

- 通常DaqOperatorは変更する必要はない。
- /usr/libexec/daqmw/DaqOperatorにバイナリがある。

開発環境

- newcomp
  - 新規コンポーネント開発開始用テンプレート作成ツール
    - C++のテンプレートではありません
- Makefile
  - あんまりぐちゃぐちゃ書かなくても済むようにしてみた。

newcomp

- newcomp MyMonitor
  - MyMonitorディレクトリを作成
  - Makefile、MyMonitor.h、MyMonitor.cpp、MyMonitorComp.cppのひな形を作る
  - インクルードガード、コンポーネント名は
    MYMONITOR、MyMonitorになる（ケースマッチで引数から取得）
  - InPort 1 個、OutPort 1 個、実装すべきメソッド（ほぼ空）のものができる（ポート名はmysource_in, mysource_out）
- Source型、Sink型はオプションがある
  - newcomp -t source MyReader
  - newcomp -t sink　　MyMonitor
  - それぞれInPort, OutPortは必要なものだけ作る。
- newcomp -h で簡単なヘルプがでる。

newcomp -h

Usage: newcomp [-c] [-f] [-t component_type] NewCompName

Create skeleton component files with NewCompName in NewCompName directory.
If this directory does not exist, it will be created automatically
unless -c option is specified.
Please specify NewCompName as you need.
If you issue "newcomp NewGatherer", following files will be created:

NewGatherer/
NewGatherer/Makefile
NewGatherer/NewGatherer.h
NewGatherer/NewGatherer.cpp
NewGatherer/NewGathererComp.cpp

You may specify component type as -t option. Valid component types are:

null
sink
source

Other option:
-c Don't create directory but create component files in the current directory
-f Overwrite exisiting file(s)

newcomp –t source MyReader

```
//MyReader.h
private:
   TimedOctetSeq          m_out_data;
   OutPort<TimedOctetSeq> m_OutPort;


// MyReader.cpp   Ctor
MyReader::MyReader(RTC::Manager* manager)
   : DAQMW::DaqComponentBase(manager),
     m_OutPort("myreader_out", m_out_data),
     m_out_status(BUF_SUCCESS),
```

newcomp –t sink MyMonitor

```
//MyMonitor.h
private:
   TimedOctetSeq         m_in_data;
   InPort<TimedOctetSeq>  m_InPort;

//MyMonitor.cpp  Ctor
MyMonitor::MyMonitor(RTC::Manager* manager)
   : DAQMW::DaqComponentBase(manager),
     m_InPort("mymonitor_in",   m_in_data),
```

newcompで入るロジック置き場

- source, sinkそれぞれ典型的な使い方はこうだろうと思ったものの空のものが入っている。
- source: read_data_from_detectors()
- sink: online_analyze()

newcomp誕生までの道のり

- 以前（おととし）まではSkeletonファイルをコピーして

```
for i in Skeleton*; do
   sed -i.bak -e 's/skeleton/mymonitor/' ¥
              -e 's/Skeleton/MyMonitor/' ¥
              -e 's/SKELETON/MYMONITOR/' $i
done
```

とかしていた。

マニュアルにもそう書いた（つもりだった）がコマンドがまちがっていました（申し訳ありません）。

- さすがに上のコマンドをいちいち実行するのはどうかしていると思って、スクリプトで実行させることにした。

newcompでできるMakefileの使い方

- 雛型が作るMakefileに
  - ソースファイルが増えたら SRCS += として追加する。
  - インクルードファイルの場所は CPPFLAGS += で追加する。
  - ライブラリファイルは

    LDLIBS += -L/path/to/lib –lmylib

    で追加する。
  - あとはincludeしているcomp.mkとimplicit ruleが面倒をみる。

newcompでできるMakefile

```
COMP_NAME = MyMonitor

all: $(COMP_NAME)Comp

SRCS += $(COMP_NAME).cpp
SRCS += $(COMP_NAME)Comp.cpp

# sample install target
#
# MODE = 0755
# BINDIR = /tmp/mybinary
#
# install: $(COMP_NAME)Comp
#       mkdir -p $(BINDIR)
#       install -m $(MODE) $(COMP_NAME)Comp $(BINDIR)

include /usr/share/daqmw/mk/comp.mk
```

Makefile一般論 (implicit rule)

- hello.cがあったらMakefileなしでも
  make hello
  でOK
- CFLAGS:
- CXXFLAGS:
- CPPFLAGS:
- LDLIBS: GNU make。＊BSDではLDADD

Makefile一般論(implicit rule, LDLIBS)

```
PROG = sample
CFLAGS = -g -O0 -Wall
LDLIBS += -lm

all: ${PROG}

clean:
	rm -f *.o ${PROG}
```

```
% make
cc -g -O0 -Wall    sample.c  -lm -o sample
```

GNU Make

- Linuxにたいてい採用されているGNU Makeだとobjectファイルを追加していくようだ

```
all: hello

OBJS += hello.o
OBJS += options.o
OBJS += help.o

hello: $(OBJS)
```

Makefile

- DAQ-Middlewareで提供するMakefileは、以前はOBJS += でオブジェクトファイル名を指定する方式だった。が、あるとき:
- ファイルが増えてきてMakefileをアップデートするときviで

  :r! ls -1 *.c

  して (あるいはlsの出力をコピーアンドペーストして)

  aaa.c

  bbb.c

  ccc.c

  ソースファイルを並べて、この先頭にOBJS += を追加。そのあと.cを.oに変更するつもりで、忘れていて

  make clean

  でソースが消えた！(というか自分で消したんだが)
- FIX: SRCS += でソースを追加する方式に変更

Makefile
自動生成されるファイルの対処

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

makeしたら自動生成でこれより多い数のソースが出現。

めざわりなので自動生成されるファイル群はautogenディレクトリへ押し込め。

DAQシステムの起動

- コンフィギュレーションファイルを書く
  - 今はまだGUIがありません（すみません）
  - サンプルをコピーして手で編集
  - だいたいここで間違いが入ることがおおいです（だからGUIがあればよいんだが。重ねてお詫び申し上げます）。ので
    /usr/share/daqmw/examples/以下にあるサンプルコンポーネントのコンフィギュレーションは全部/usr/share/daqmw/conf/に入れた。
- システム統括はDaqOperatorが行いますが、各コンポーネントは既に起動している必要があります
- コンポーネントの起動方法
  - 手でコマンドラインから起動
  - ネットワークブート
  - コンフィギュレーションファイルにexecPathがあるからこれを読んでプログラムが起動 (run.pyの目的その1）

run.py

- 開発中は
  - DaqOperatorをコンソールモードで
  - 各コンポーネントはlocal計算機で

  起動することが多いかと思うのでここではこの方法だけを扱います

コマンド:　　run.py –c –l config.xml

-c: console modeでDaqOperatorを起動

-l: ローカル計算機で各コンポーネントを起動

run.py –c –l config.xml 動作説明

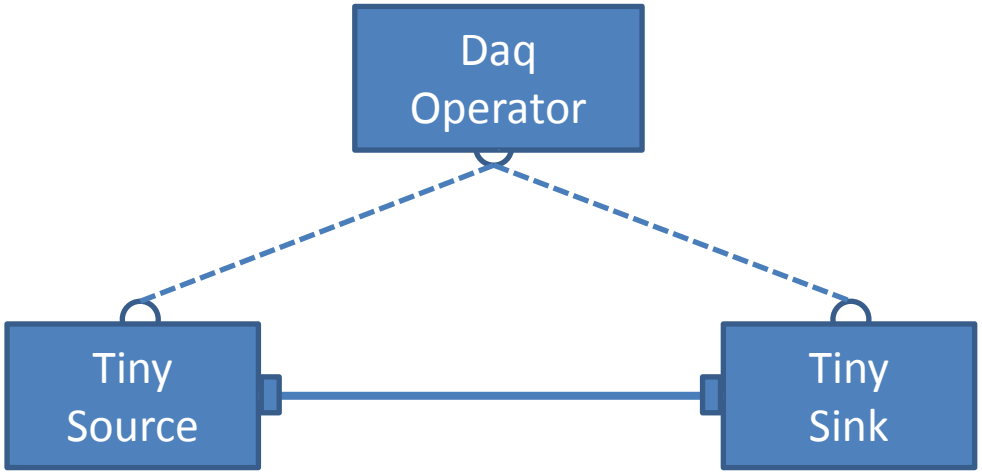
- xmllintで引数で指定されたconfig.xmlのvalidationを実行 (config.xmlのスキーマは
  /usr/share/daqmw/conf/config.xsd)
- ネームサーバーの起動
- config.xml内のexecPathからコンポーネントパス名を取得してそれらを起動
- 最後にDaqOperatorをコンソールモードで起動し、run.pyはDaqOperatorが終了するのを待つ。
- コンソールモードで起動したDaqOperatorの動作:
  - コンソールモードで起動したDaqOperatorへの支持は端末(コンソール)経由でキーボードから手入力 (httpではない)
  - DaqOperatorはコンソールモードで起動すると端末に各コンポーネントが扱ったバイト数を表示

開発マニュアルでの例題

- いずれもソース、コンフィギュレーションファイルは/usr/share/daqmw/examples/, /usr/share/daqmw/conf/の下にあります。

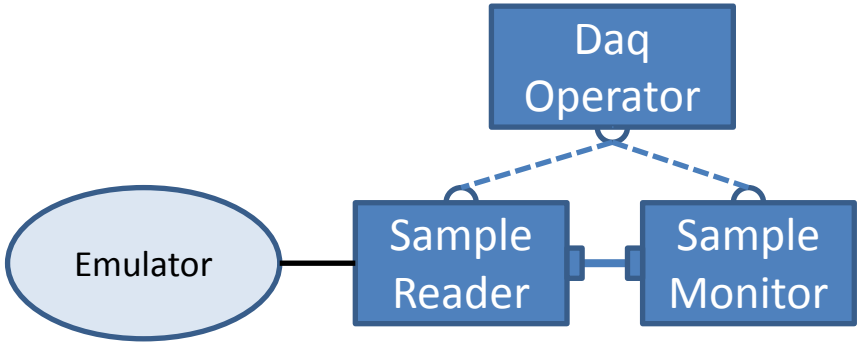
- Skeletonコンポーネントでの状態遷移の確認 (23ページ)
- コンポーネント間のデータ通信 (27ページ)
- エミュレータからのデータを読んでROOTでヒストグラムを書くシステムの開発 (31ページ)
- 上のシステムのコンディションデータベース化(58ページ)

コンポーネント間のデータ通信 (27ページ)

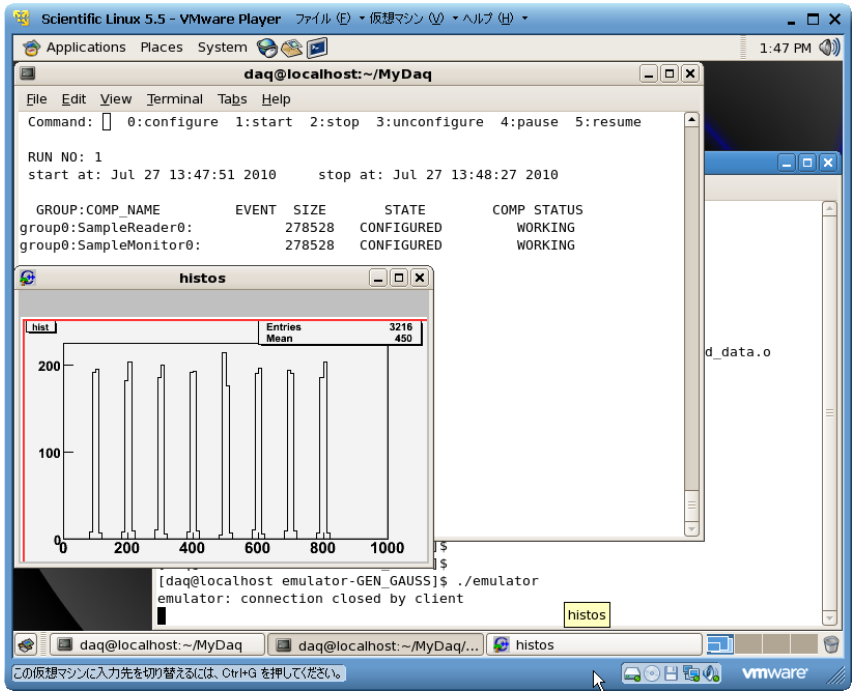


- TinySourceは適当に数値を入れておくる
- TinySinkは受け取ったデータを標準エラーに出力する
- run.py –cl tiny.xml で起動したコンポーネントのエラーログは /tmp/daqmw/log.CompName（CompNameはコンポーネント名）に出力される(TinySinkのログは/tmp/daqmw/log.TinySinkに出力される）

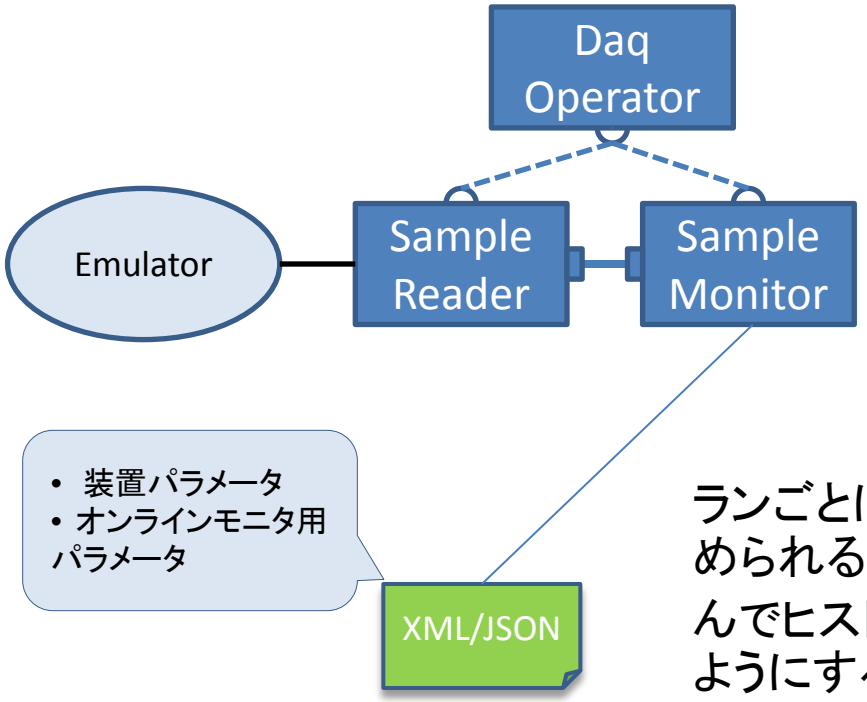
SampleReader, SampleMonitor



- Emulatorからのデータを読んでROOTでヒストグラムを書く



Conditionデータベース



ランごとにヒストグラムパラメータを決められるようにdaq_start()でこれを読んでヒストグラムのパラメータを決めるようにする

デモ

データソースの準備

- Emulatorを作るとか実機を用意するとか
- 今回はemulatorを使います。

/home/daq/MyDaq/emulator-GEN_GAUSS

Emulatorの仕様

- ./emulator [-t tx_bytes/s] [-b buf_bytes] [-h ip_address]
- デフォルトは –t 8k –b 1k (8kB/sec, 1回1kb)
- 数値はm, kのサフィックスが使える
- 指定された転送レートをできるだけ守るようにデータを送る
- 送ってくるデータフォーマット:

Magic	Format Version	Module Number	Reserved	Event Data	Event Data	Event Data	Event Data

Magic: 0x5a
Format Version: 0x01
Module Number: 0x00 – 0x07
Event Data: 適当にガウシャン風。100, 200, 300, ... 800にピークがある。
1000倍した整数値で送ってくる。ネットワークバイトオーダー。

Emulatorの転送レートアルゴリズム

- cstream
  http://www.cons.org/cracauer/cstream.html
  のアルゴリズムをそのまま使用
- スタート時刻をgettimeofday()で取得
- 一定バイト数write()
- 書き終わったらgettimeofday()で時刻を取得
- スタート時刻からの時間経過がわかるのでこれまでの転送レートがわかる
- 書きすぎだったら、どのくらいsleep()したら指定された転送レートに合わせられるか計算できるのでそのぶんsleep()する
- 書きすぎてなかったらsleep()なしにwrite()する。

Emulatorの注意

- 指定された（あるいはデフォルトの）転送レートを守るように作ったのでどの実験のデータフローともまったく異なったデータフローになっているはずで実用の意味はあまりないと思う。

デモ (1)

- 起動して nc で読んでみる

  cd /home/daq/MyDaq/emulator-GEN_GAUSS

  ./emulator

  別の端末で

  nc localhost 2222 > data

  数秒後Ctrl-Cで停止させて

  hexdump –vC data

  でダンプして中身をみる。

デモ (2)
SampleReader, SampleMonitor

- cd ~/MyDaq
- cp -r /usr/share/daqmw/examples/SampleReader .
- cp -r /usr/share/daqmw/examples/SampleMonitor .
- cd SampleReader
- make
- cd ..
- cd SampleMonitor
- cd ..
- cp /usr/share/daqmw/conf/sample.xml
- run.py -cl sample.xml

Web UI

- SampleReaderとSampleMonitorをWeb UI (DAQ-Middlewareで配布しているサンプル実装)で動かす。
- (註) Web ブラウザは現在のところ Firefoxに限り動作する。