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
  - 1.1.0が最新

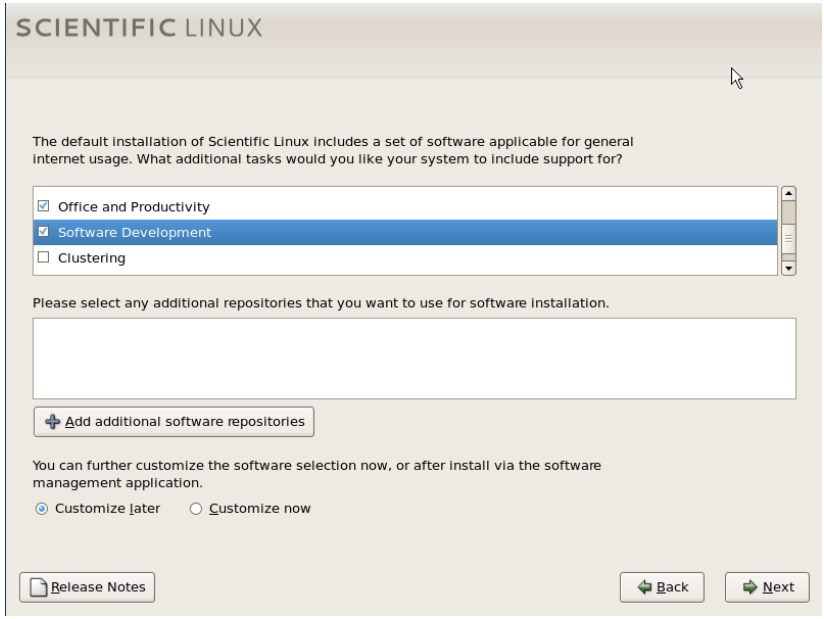
  http://daqmw.kek.jp/docs/DAQ-Middleware-1.1.0-Tech.pdf

- DAQ-Middleware 1.4.0開発マニュアル

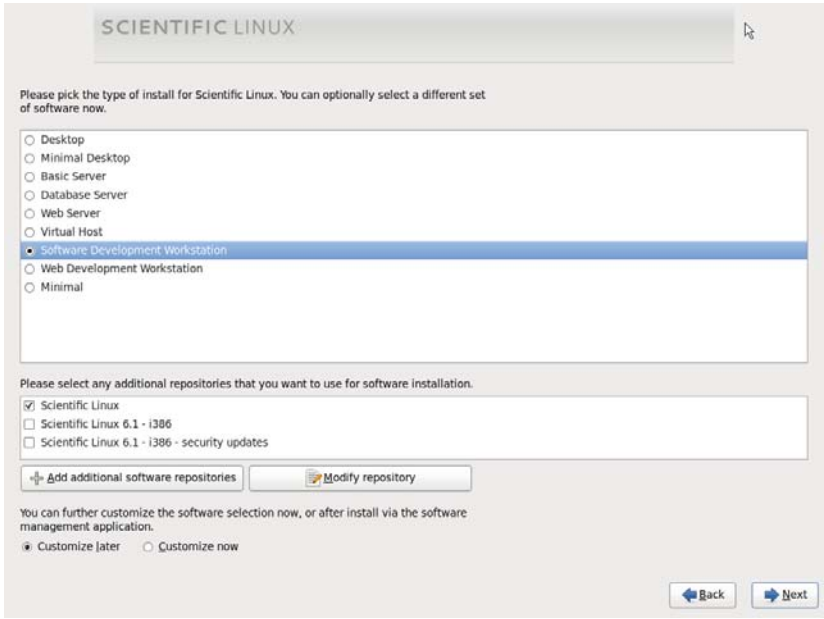
  http://daqmw.kek.jp/docs/DAQ-Middleware-1.4.0-DevManual.pdf

OS セットアップ

- Scientific Linuxの場合は"Software Development"を選択しておく



Scientific Linux 5



Scientific Linux 6

OS セットアップ トラブル解決法 Scientific Linux編

- DAQコンポーネントのコンパイルで libuuidがない（/usr/bin/ld: cannot find -luuid）と言われたら
  - yum install e2fsprogs-devel (SL 5.x)
    (Software developmentを選ぶと自動で入る)
  - yum install libuuid-devel (SL 6.x)
    (Software Developmentを選んでも自動で入らない）
- iptables (packet filtering firewall) をオフ
  - chkconfig iptables off; service iptables stop
- WebUIを使うときはSELinuxをオフ
  - Scientific Linux 5.xではインストール終了後、最初のブートでダイアログがでる。SL 6では出ない。
  - /etc/sysconfig/selinuxで SELINUX=enforcing → SELINUX=disabledに書き換えてリブート
  - 本来はSELinuxをオフにしなくてもよいはずだが

DAQ-Middlewareセットアップ Scientific Linux

- Scientific Linux (CentOS, RHEL) 5.x、6.xの場合は

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

  % su

  root# sh daqmw-rpm install

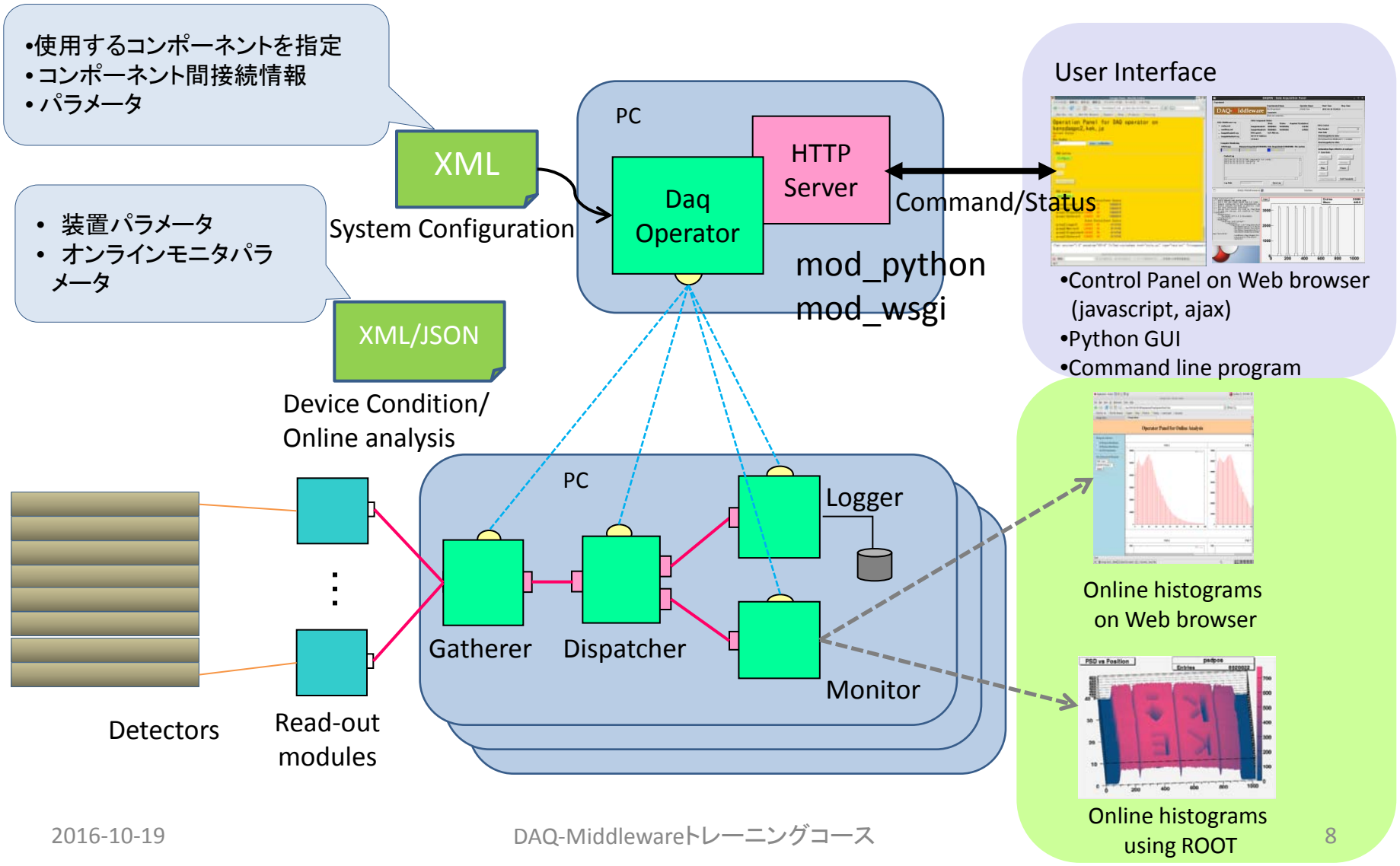
  ファイル一覧: rpm -ql DAQ-Middleware

  アンインストールは sh daqmw-rpm uninstall
- 配布しているVirtualBoxイメージは上のコマンドで作成

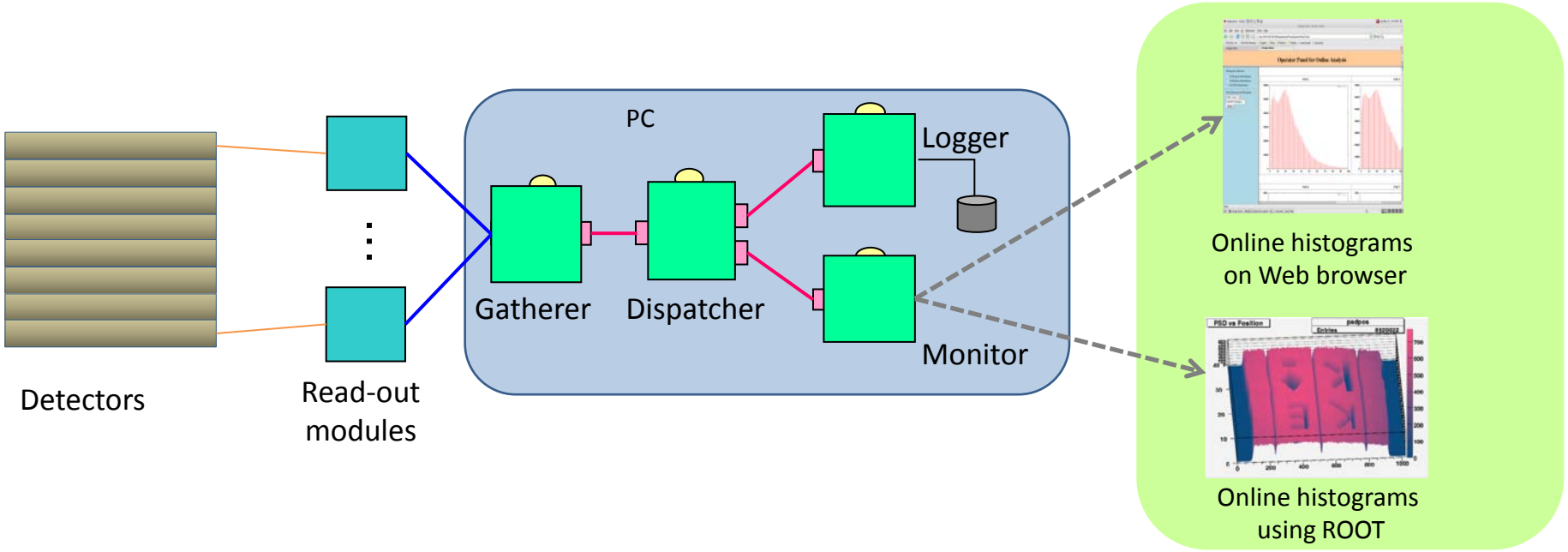
DAQ-Middlewareセットアップ Ubuntu 2012.04 LTS

- wget http://daqmw.kek.jp/src/Ubuntu_daqmw
  sudo sh Ubuntu_daqmw install

DAQ-Middleware構成図



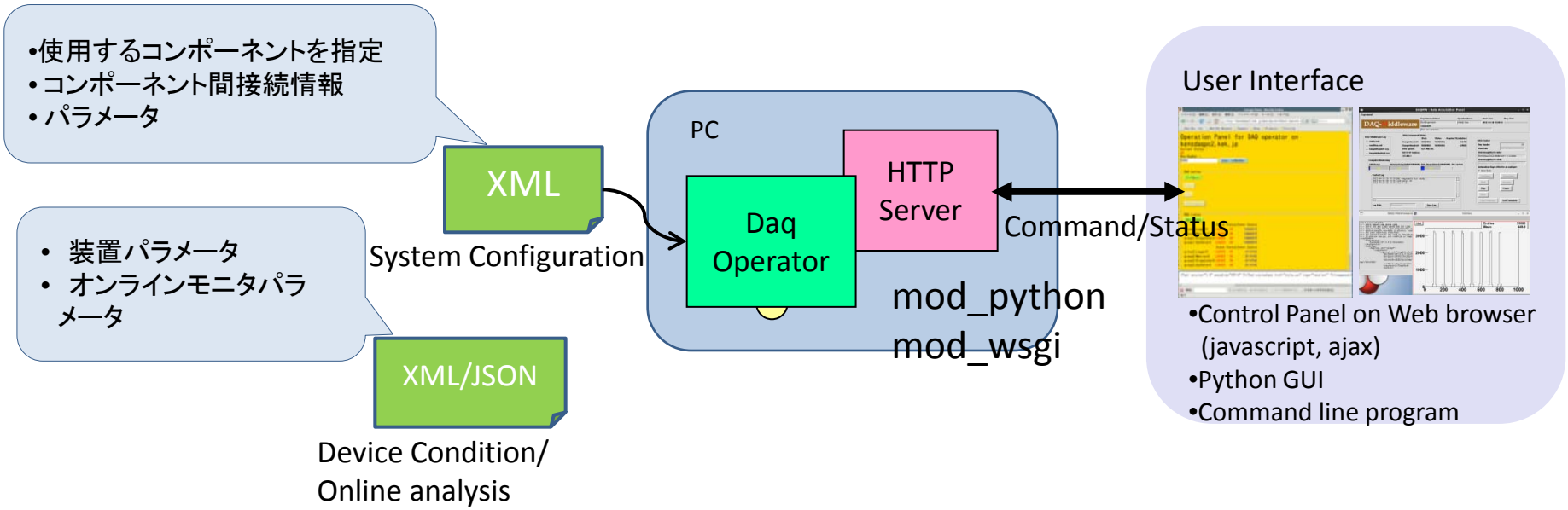
データ収集パス



複数のDAQコンポーネントを組み合わせてデータ収集パスを作る。

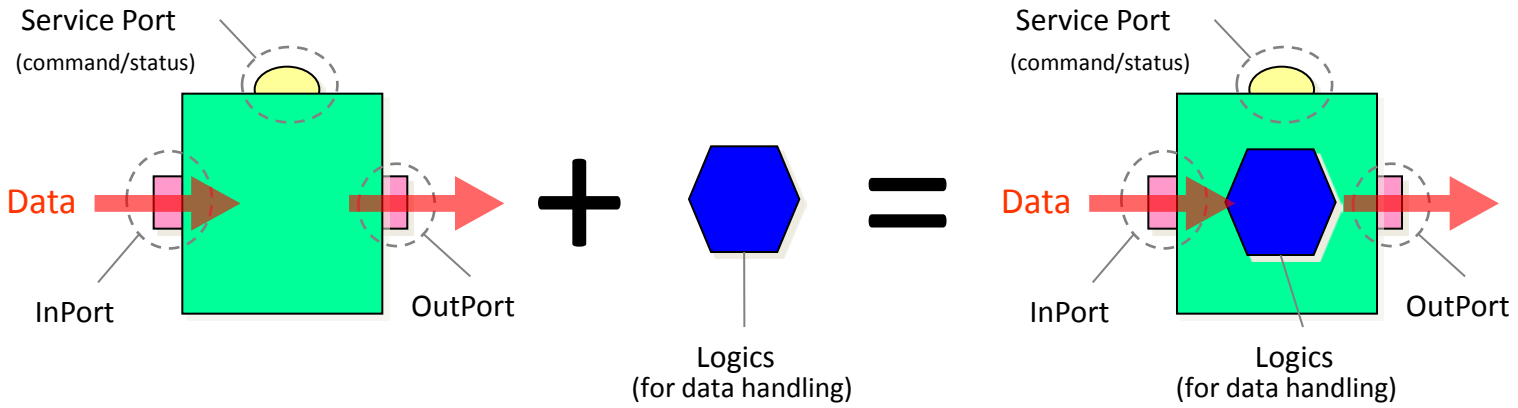
- DAQ-Middlewareで提供するパス（ネットワーク接続）
- リードアウトモジュール - gatherer間はネットワークだったり
  その他だったりする（リードアウトモジュールによる）

ランコントロール



- DaqOperator: DAQコンポーネントを統括する
- DaqOperatorに対する指示はhttpで行う
  - •既存のものがあるときはそれがhttpで通信するようにすれば使える

DAQコンポーネント

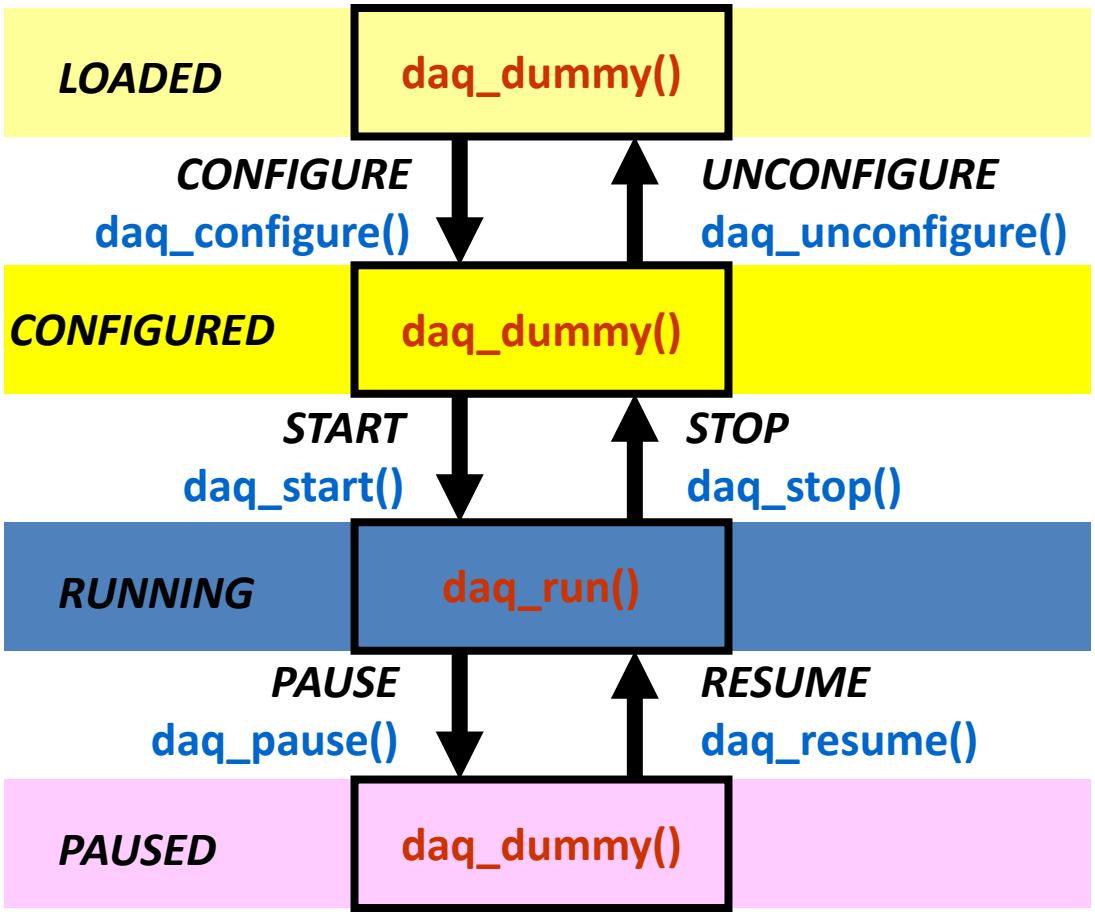


- DAQコンポーネントを組み合わせてDAQシステムを構築する。
- 上流からのデータを読むにはInPortを読む。
- データを下流に送るにはOutPortに書く。
- DAQコンポーネント間のデータ転送機能はDAQ-Middlewareが提供する
- ユーザーはコアロジックを実装することで新しいコンポーネントを作成できる。
  コアロジックの例:
  - リードアウトモジュールからのデータの読み取りロジック
  - ヒストグラムの作成ロジック

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

コンポーネント状態遷移



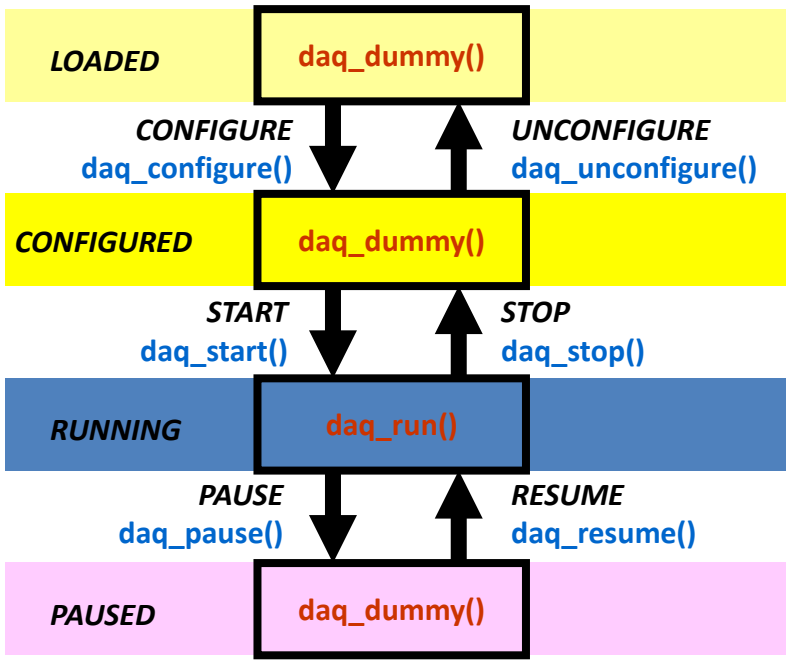
各状態(LOADED, CONFIGURED, RUNNING, PAUSED)にある間、対応する関数が繰り返し呼ばれる。

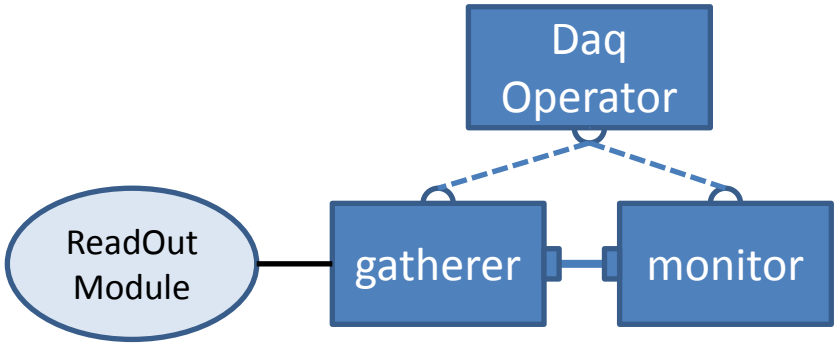
状態遷移するときは状態遷移関数が呼ばれる。

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）

各関数を実装することでDAQコンポーネントを完成させる。

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

1コンポーネントに必要なソースファイル

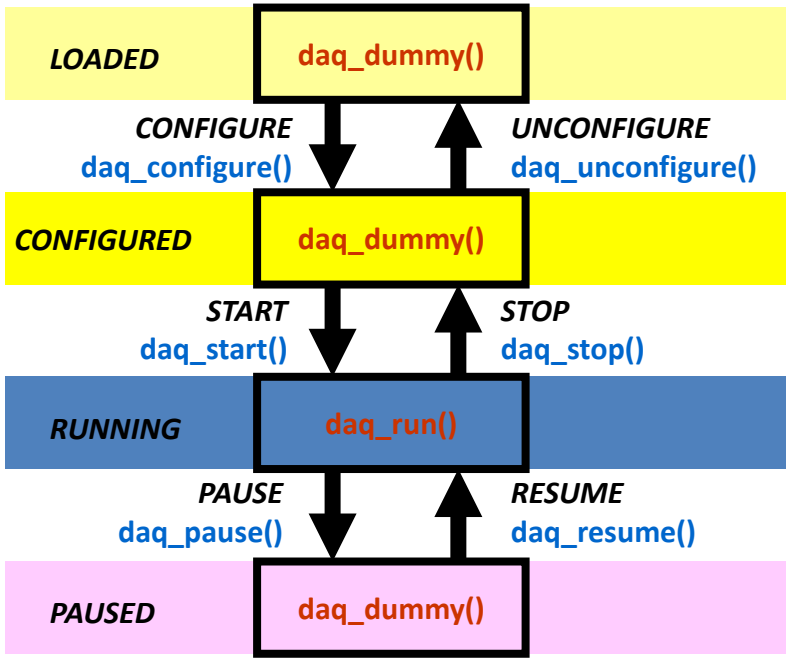
Skeletonという名前のコンポーネントの場合

- 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。変更が必要ない場合が多い)
- Makefile
- その他分離したくなったファイル
  - 例：Monitorのイベントデータデコード関数
    Loggerのファイルopen, closeの部分

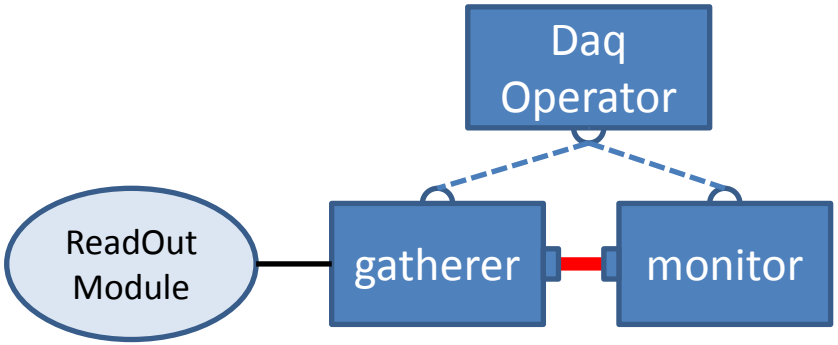
コンポーネント実装方法

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

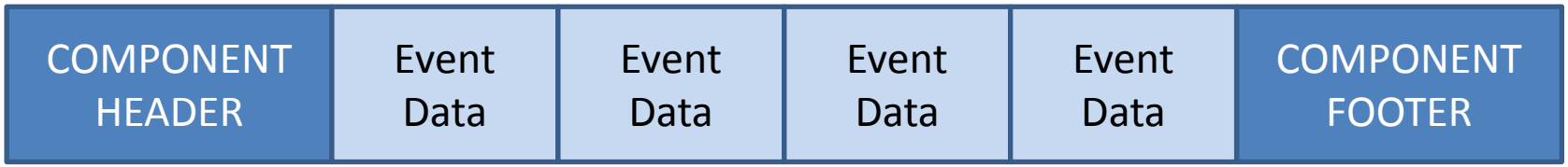
- daq_configure()
- daq_start()
- daq_run()
- daq_stop()
- daq_unconfigure()
- daq_pause()



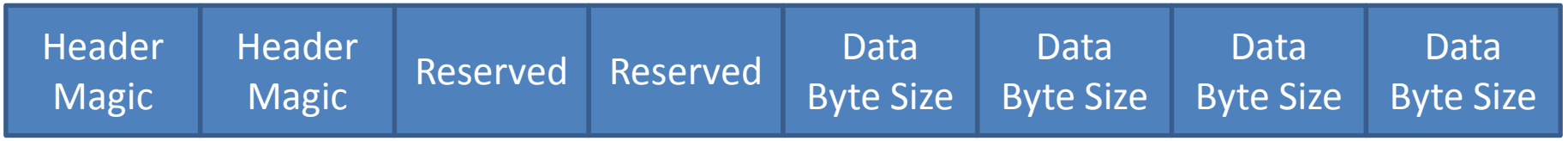
コンポーネント間のデータフォーマット



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イベント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（全部で4バイト）はユーザが使用してもよい。
例：イベントデータフォーマットにモジュール番号を入れるところがないのでそれを入れるなど。

ReservedだとDAQ-Middleware側で今後なにかに使うような意味になるのでUserData等適当な言葉に置き換えるかもしれません。

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)

- check_header(unsigned char *header, unsigned received_byte)
- check_footer(unsigned char *footer)
- 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

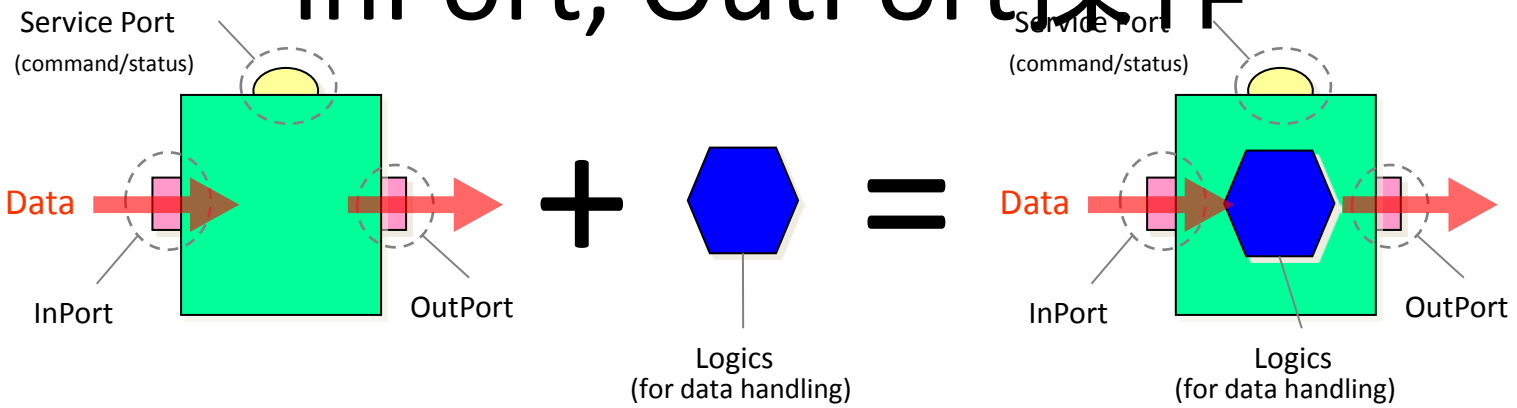
エラー時の対処

- 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。
  - fatal_error_report()内で例外がthrowされる
- 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;
```

m_in_dataという識別子を定義したのみ。
InPortのデータを扱うハンドラにはまだ
なっていない。

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配列に書く
  （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。

DaqOperatorからコンポーネントにパラメータを送る

- config.xmlの<params>に書くとそのパラメータがconfigure時に各コンポーネントに送られる
- 各コンポーネントにはNVListで送られるので送られてきたものを順次読む
  - NVList

    Name0, Value0, Name1, Value1, Name2, Name2, ...

    全てstringで送られているので数値に変更するにはstrtol() (あるいは古くはatoi()) を使って変換する。

```
<! -- sample.xml -->
<components>
    <component cid="SampleReader0">
        :
        :
        <params>
            <param pid="srcAddr">127.0.0.1</param>
            <param pid="srcPort">2222</param>
        </params>
```

新規に開発するには

- 似たようなものから改造する
  - SampleReader (リードアウトモジュールからネットワーク経由で読みとり)
  - SampleMonitor (ROOTでヒストグラムを作って画面に表示する)
  - SampleLogger (ディスクにデータを保存。1GBおきに別ファイルに保存する)
- newcompコマンドを使ってファイルを作りなかみは一から実装
- 似たようなものから改造するのがほとんど

Makefile (Sample*)

```
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の使い方

- 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ディレクトリは 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。

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できる。

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
  - 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
    /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/にある。
  - GUI: confPanel.py
- 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

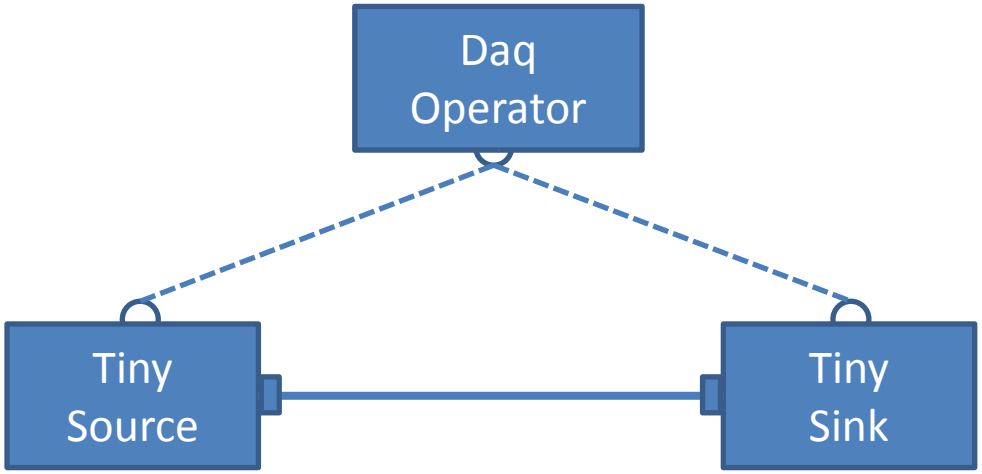
- xmllintで引数で指定されたconfig.xmlのvalidationを実行 (config.xmlのスキーマは
  /usr/share/daqmw/conf/config.xsd)
- ネームサーバーの起動
- config.xml内のexecPathからコンポーネントパス名を取得してそれらを起動
- 最後にDaqOperatorをコンソールモードで起動し、run.pyはDaqOperatorが終了するのを待つ。
- コンソールモードで起動したDaqOperatorの動作:
  - コンソールモードで起動したDaqOperatorへの支持は端末(コンソール)経由でキーボードから手入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

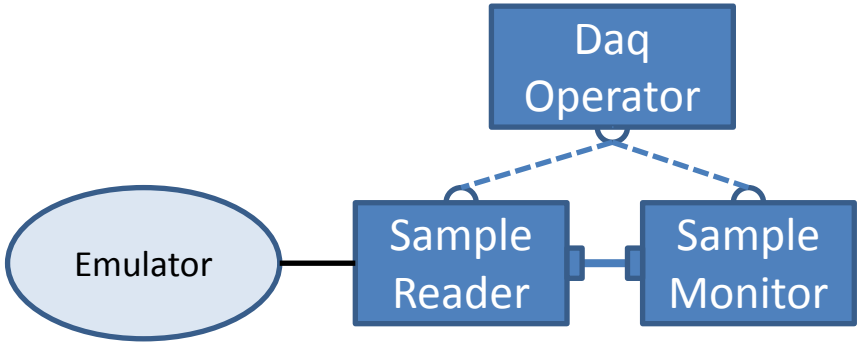
- Skeletonコンポーネントでの状態遷移の確認 (25ページ)
- コンポーネント間のデータ通信 (29ページ)
- エミュレータからのデータを読んでROOTでヒストグラムを書くシステムの開発 (33ページ)
- 上のシステムのコンディションデータベース化(60ページ)

コンポーネント間のデータ通信 (29ページ)

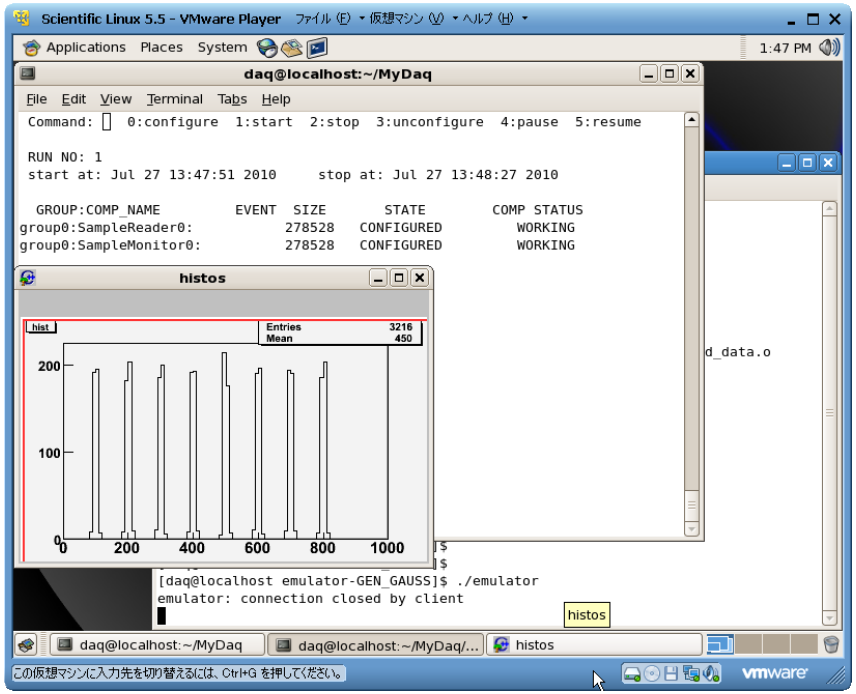


- TinySourceは適当に数値を入れておくる
- TinySinkは受け取ったデータを標準エラーに出力する
- run.py –cl tiny.xml で起動したコンポーネントのエラーログは /tmp/daqmw/log.CompName（CompNameはコンポーネント名）に出力される(TinySinkのログは/tmp/daqmw/log.TinySinkに出力される）

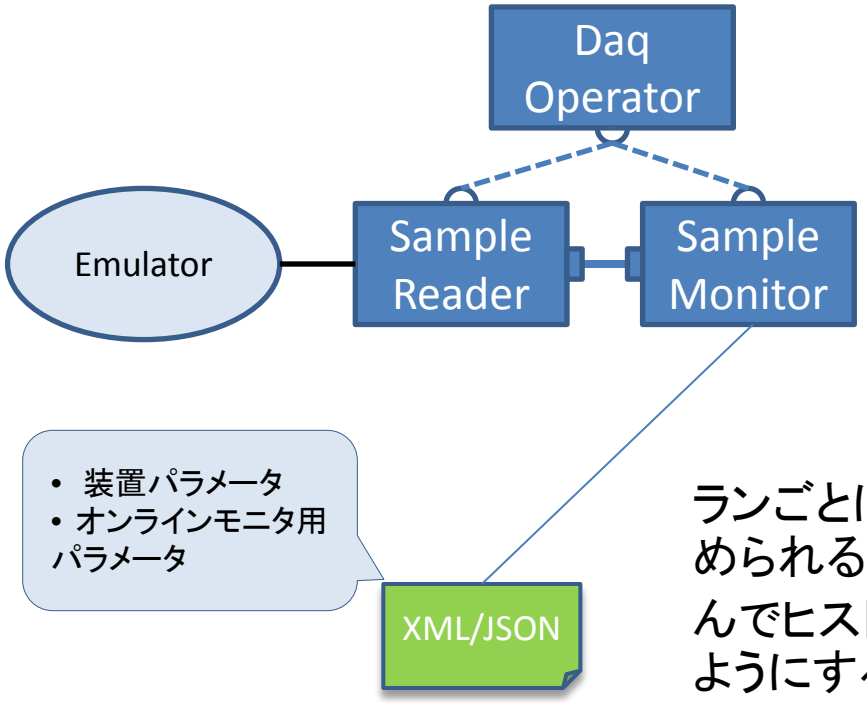
SampleReader, SampleMonitor



- 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

データソースの準備

- Emulatorを作るとか実機を用意するとか
- ここではDAQ-Middleware付属のemulatorを使います。
- /usr/bin/daqmw-emulator

Emulatorの仕様

- daqmw-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

  daqmw-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。

```
% hexdump -vC data | head
00000000  5a 01 00 00 00 01 91 03  5a 01 01 00 00 03 0f 8f  |Z.......Z.......|
00000010  5a 01 02 00 00 04 93 0a  5a 01 03 00 00 06 39 8f  |Z.......Z.....9.|
00000020  5a 01 04 00 00 07 b1 8c  5a 01 05 00 00 09 27 4f  |Z.......Z.....'O|
00000030  5a 01 06 00 00 0a 96 e1  5a 01 07 00 00 0c 25 fb  |Z.......Z.....%.|
00000040  5a 01 00 00 00 01 9b 46  5a 01 01 00 00 03 13 62  |Z......FZ......b|
00000050  5a 01 02 00 00 04 b3 a8  5a 01 03 00 00 06 16 cf  |Z.......Z.......|
00000060  5a 01 04 00 00 07 85 c2  5a 01 05 00 00 09 27 52  |Z.......Z.....'R|
00000070  5a 01 06 00 00 0a a5 e1  5a 01 07 00 00 0c 44 cd  |Z.......Z.....D.|
00000080  5a 01 00 00 00 01 63 fb  5a 01 01 00 00 03 06 2a  |Z.....c.Z......*|
00000090  5a 01 02 00 00 04 a0 d9  5a 01 03 00 00 06 1a c0  |Z.......Z.......|
```

デモ (2)
SampleReader, SampleMonitor

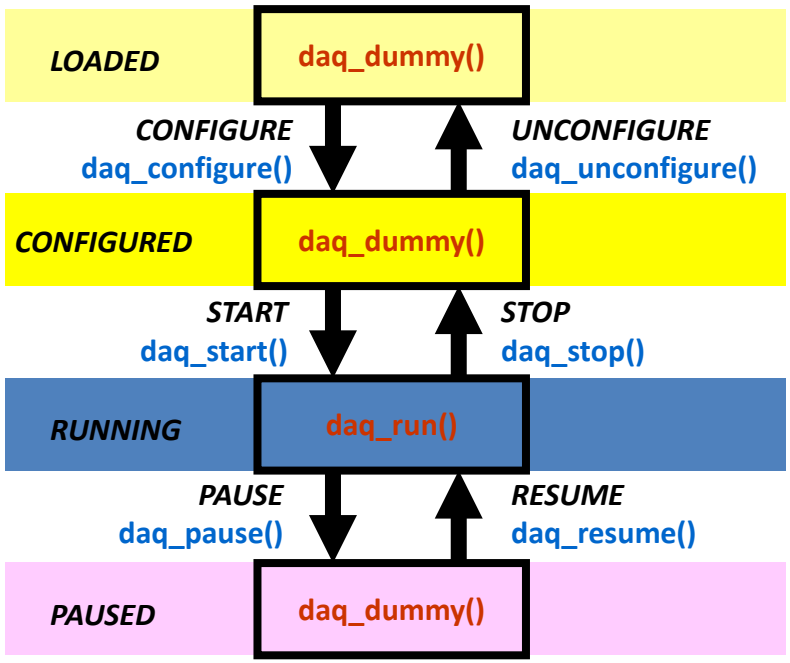
- cd ~
- mkdir -p MyDaq (-p: なければ作る)
- cd ~/MyDaq
- cp -r /usr/share/daqmw/examples/SampleReader .
- cp -r /usr/share/daqmw/examples/SampleMonitor .
- cd SampleReader
- make
- cd ..
- cd SampleMonitor
- cd ..
- cp /usr/share/daqmw/conf/sample.xml
- run.py -cl sample.xml

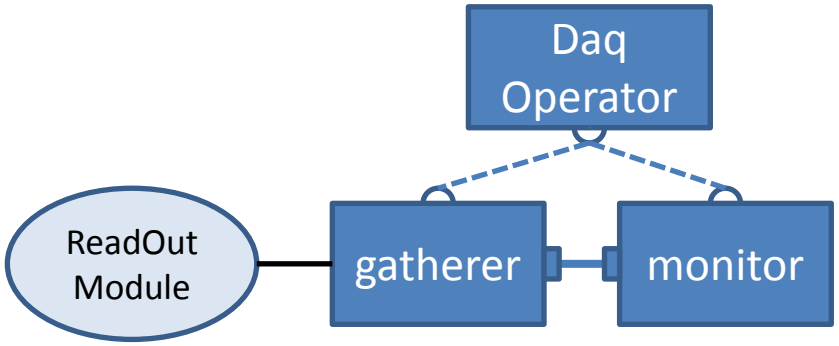
Web UI

- 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。

SampleReader, Monitorのコード解説

SampleReader, Monitorの仕様





Gatherer (SampleReader)

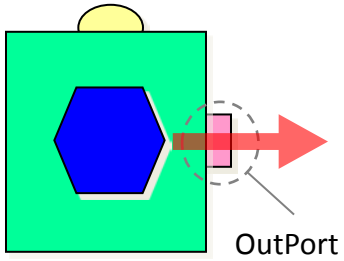
daq_configure(): リードアウトモジュールのIPアドレス、ポートを取得（DAQ-Operatorからふってくる）
daq_start(): リードアウトモジュールに接続
daq_run(): リードアウトモジュールからデータを読んで後段コンポーネントにデータを送る
daq_stop(): リードアウトモジュールから切断。

Monitor (SampleMonitor)

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

SampleReader （SampleReader.h、cpp)

```
// SampleReader.h
class SampleReader
    : public DAQMW::DaqComponentBase
{
private:
    TimedOctetSeq           m_out_data;
    OutPort<TimedOctetSeq> m_OutPort;
```




```
// SampleReader.cpp
SampleReader::SampleReader(RTC::Manager* manager)
    : DAQMW::DaqComponentBase(manager),
      m_OutPort("samplereader_out", m_out_data),
      m_sock(0),
      m_recv_byte_size(0),
      m_out_status(BUF_SUCCESS),
```

SampleReader (SampleReader.cpp) daq_configure() パラメータの取得

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

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

    return 0;
}
```

SampleReader - daq_configure()

```
<!-- config.xml -->
<params>
    <param pid="srcAddr">127.0.0.1</param>
    <param pid="srcPort">2222</param>
</params>
```

```
int SampleReader::parse_params(::NVList* list)
{
    int len = (*list).length();
    for (int i = 0; i < len; i+=2) {
        std::string sname  = (std::string)(*list)[i].value;
        std::string svalue = (std::string)(*list)[i+1].value;
        if ( sname == "srcAddr" ) {
            m_srcAddr = svalue;
        }
        if ( sname == "srcPort" ) {
            char* offset;
            m_srcPort = (int)strtol(svalue.c_str(), &offset, 10);
        }
    }
```

SampleReader - daq_start()

```
int SampleReader::daq_start()
{
    m_out_status = BUF_SUCCESS;

    // リードアウトモジュールに接続
    try {
        // Create socket and connect to data server.
        m_sock = new DAQMW::Sock();
        m_sock->connect(m_srcAddr, m_srcPort);
    } catch (DAQMW::SockException& e) {
        std::cerr << "Sock Fatal Error : " << e.what() << std::endl;
        fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
    } catch (...) {
        std::cerr << "Sock Fatal Error : Unknown" << std::endl;
        fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
    }
```

SampleReader - daq_run()

```
int SampleReader::daq_run()
{
    if (check_trans_lock()) {  // check if stop command has come
        set_trans_unlock();    // transit to CONFIGURED state
        return 0;
    }

    if (m_out_status == BUF_SUCCESS) {   // previous OutPort.write() successfully done
        int ret = read_data_from_detectors();
        if (ret > 0) {
            m_recv_byte_size = ret;
            set_data(m_recv_byte_size); // set data to OutPort Buffer
        }
    }

    if (write_OutPort() < 0) {
        ;     // Timeout. do nothing.
    }
    else {    // OutPort write successfully done
        inc_sequence_num();                      // increase sequence num.
        inc_total_data_size(m_recv_byte_size);  // increase total data byte size
    }

    return 0;
}
```

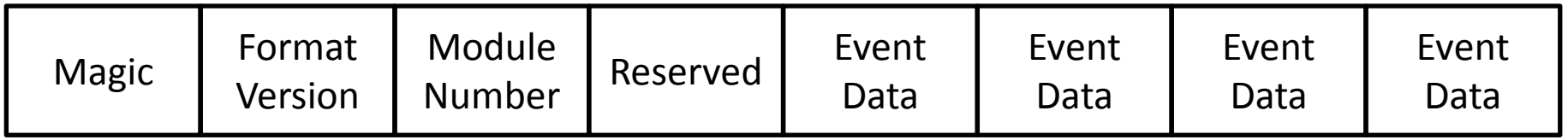
SampleReader - daq_run()

```
int SampleReader::read_data_from_detectors()
{
    int received_data_size = 0;

    /// read 1024 byte data from data server
    int status = m_sock->readAll(m_data, SEND_BUFFER_SIZE);
    // 書き方はいろいろあるがここでは先にエラーチェックを書いた
    if (status == DAQMW::Sock::ERROR_FATAL) {
        std::cerr << "### ERROR: m_sock->readAll" << std::endl;
        fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
    }
    // ここではデータがタイムアウトで読めなかったらエラーとなるように決めた
    else if (status == DAQMW::Sock::ERROR_TIMEOUT) {
        std::cerr << "### Timeout: m_sock->readAll" << std::endl;
        fatal_error_report(USER_DEFINED_ERROR2, "SOCKET TIMEOUT");
    }
    else {
        received_data_size = SEND_BUFFER_SIZE;
    }

    return received_data_size;
}
```

SampleMonitor - SampleData.h

```
#ifndef SAMPLEDATA_H
#define SAMPLEDATA_H

const int ONE_EVENT_SIZE = 8;

struct sampleData {
  unsigned char magic;
  unsigned char format_ver;
  unsigned char module_num;
  unsigned char reserved;
  unsigned int  data;
};

#endif
```

データフォーマット構造体を定義。
デコードしたらすぐにこの構造体に代入して、変数名で処理できるようにする。



SampleMonitor.h

```
////////// ROOT Histogram //////////
    TCanvas *m_canvas;
    TH1F    *m_hist;
    int      m_bin;
    double   m_min;
    double   m_max;
    int      m_monitor_update_rate;
    unsigned char m_recv_data[4096];
    unsigned int  m_event_byte_size;
    struct sampleData m_sampleData;

    bool m_debug;
};
```

SampleMonitor.cpp - daq_dummy()

```
int SampleMonitor::daq_dummy()
{
    if (m_canvas) {
        m_canvas->Update();
        // daq_dummy() will be invoked again after 10 msec.
        // This sleep reduces X servers' load.
        sleep(1);
    }

    return 0;
}
```

SampleMonitor - daq_configure()

```
int SampleMonitor::daq_configure()
{
    ::NVList* paramList;
    paramList = m_daq_service0.getCompParams();
    parse_params(paramList);

    return 0;
}

int SampleMonitor::parse_params(::NVList* list)
{
    int len = (*list).length();
    for (int i = 0; i < len; i+=2) {
        std::string sname  = (std::string)(*list)[i].value;
        std::string svalue = (std::string)(*list)[i+1].value;

        std::cerr << "sname: " << sname << "  ";
        std::cerr << "value: " << svalue << std::endl;
    }

    return 0;
}
```

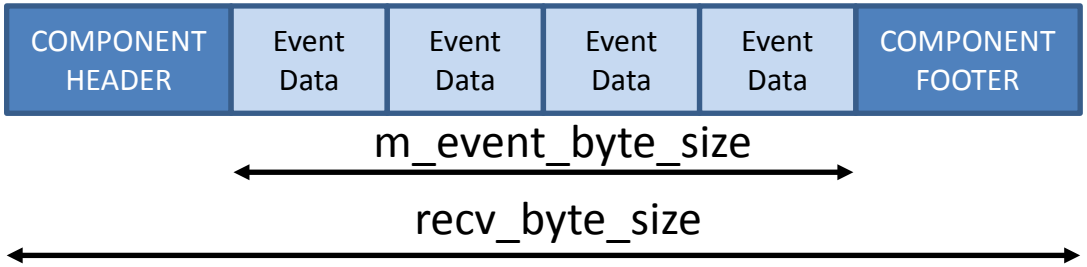
config.xml
SampleMonitorのparamsは空なのでなにもしていない。

SampleMonitor - daq_start()

```
int SampleMonitor::daq_start()
{
    m_in_status  = BUF_SUCCESS;

    ///////////////// CANVAS FOR HISTOS ///////////////////
    if (m_canvas) {
        delete m_canvas;
        m_canvas = 0;
    }
    m_canvas = new TCanvas("c1", "histos", 0, 0, 600, 400);

    ////////////////       HISTOS      ///////////////////
    if (m_hist) {
        delete m_hist;
        m_hist = 0;
    }

    int m_hist_bin = 100;
    double m_hist_min = 0.0;
    double m_hist_max = 1000.0;

    m_hist = new TH1F("hist", "hist", m_hist_bin, m_hist_min, m_hist_max);

    return 0;
}
```

ヒストグラムデータ生成

SampleReader - daq_run()



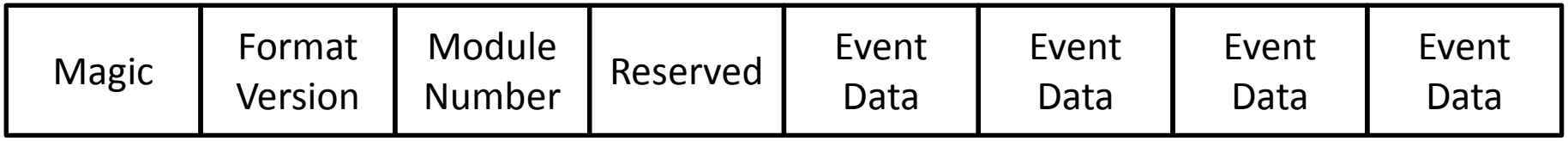

```
int SampleMonitor::daq_run()
{
    unsigned int recv_byte_size = read_InPort();
    if (recv_byte_size == 0) { // Timeout 読むデータがなかった
        return 0;
    }

    check_header_footer(m_in_data, recv_byte_size); // check header and footer
    m_event_byte_size = get_event_size(recv_byte_size);

    //////////////  Write component main logic here. //////////////
    memcpy(&m_recv_data[0], &m_in_data.data[HEADER_BYTE_SIZE], m_event_byte_size);

    fill_data(&m_recv_data[0], m_event_byte_size);

    if (m_monitor_update_rate == 0) {
        m_monitor_update_rate = 1000;
    }

    unsigned long sequence_num = get_sequence_num();
    if ((sequence_num % m_monitor_update_rate) == 0) {
        m_hist->Draw();
        m_canvas->Update();
    }
```

SampleMonitor - fill_data()

```
int SampleMonitor::fill_data(const unsigned char* mydata, const int size)
{
    for (int i = 0; i < size/(int)ONE_EVENT_SIZE; i++) {
        decode_data(mydata);
        float fdata = m_sampleData.data/1000.0; // 1000 times value is received
        m_hist->Fill(fdata);

        mydata+=ONE_EVENT_SIZE;
    }
    return 0;
}
```

SampleMonitor - decode_data()

```
int SampleMonitor::decode_data(const unsigned char* mydata)
{
    m_sampleData.magic      = mydata[0];
    m_sampleData.format_ver = mydata[1];
    m_sampleData.module_num = mydata[2];
    m_sampleData.reserved   = mydata[3];
    unsigned int netdata    = *(unsigned int*)&mydata[4];
    m_sampleData.data       = ntohl(netdata);
```



ntohl(): 複数バイトの数値を送るときに、桁が大きいほうが先にくる流儀と
桁が小さいほうが先にくる流儀がある。