DAQ-Middleware 1.1.0 技術解説書

仲吉一男

KEK 素核研

2011 年 6 月

概 要

この文書はユーザーが DAQ コンポーネントを開発しデータ収集システムを構築する際に必要となる **DAQ-Middleware 1.1.0** の仕様を解説します。また新規に DAQ ミドルウェアの開発に加わる開発者がその仕様を理解するのに役にたてばと思っています。第 2 節で DAQ ミドルウェアのアーキテクチャの概要を説明し、第 3 節で DAQ コンポーネントの仕様について説明し、第 4 節で DAQ オペレータの仕様を説明します。

目 次

1　はじめに

この文書はユーザーが DAQ コンポーネントを開発する際に必要となる **DAQ-Middleware 1.1.0** の仕様について技術的な解説をするために書かれました。**DAQ-Middleware 1.1.0**　の仕様の技術的な解説の前にその概要を説明します。DAQ(Data AQuisition) 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・コンポーネントを組み合わせて DAQ システムを構築します。DAQ ミドルウエアの実装は、RT(Robot Technology) ミドルウェア [1] 技術をベースにしています。RT ミドルウエアは、独立行政法人産業技術総合研究所 (AIST) により研究・開発が行われています。RT ミドルウェアは様々なロボット要素（RT コンポーネント）を通信ネットワークを介して自由に組み合わせることで、ロボットシステムの構築を可能にするネットワーク分散コンポーネント化技術による共通プラットフォームです。RT ミドルウエアの基本ソフトウェアである RT コンポーネントは、ソフトウエアの国際標準化団体 OMG(Object Management Group) で標準仕様が採択され国際標準規格”Robotic Technology Component Specification”[2] となりました。我々は、RT コンポーネントをベースとするロボット・ネットワーク分散モデルは、データ収集にも適用可能であると考え 2006 年から AIST と共同研究を開始し、その実現可能性の検討を行ってきました [3]。その結果、RT コンポーネントに一部拡張を行うことでデータ収集においても適用可能であるという結論に至りました。RT ミドルウェアは、OMG による国際標準化後も開発が続いています。この文書では、RT ミドルウェアの産総研による実装である **OpenRTM-aist-1.0.0** (C++版）を基に開発された **DAQ-Middleware 1.1.0** (以降 DAQ ミドルウェア) について説明します。

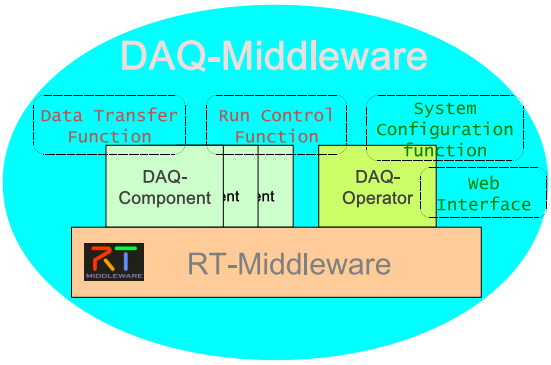

図 1: DAQ ミドルウェアと RT ミドルウェアの関係

図 1 に RT ミドルウェアと DAQ ミドルウェアの関係を表わした概念図を示します。DAQ コンポーネントは前述の RT コンポーネントを拡張して設計されています。すなわち、(1) データ収集に必要なコマンドによる状態遷移の実装、(2)RT コンポーネントのサービスポートを利用したコマンド/ステータス送受信機能の実装です。RT コンポーネントの持つ機能の一つであるデータ入出力ポートにより DAQ コンポーネント間でデータ転送を行います。DAQ オペレータは、DAQ コンポーネントを制御するためのコントローラで DAQ ミドルウェア独自のもので **OpenRTM-aist-1.0.0** にはありません。ユーザから「スタート」や「ストップ」というコマンドを受け、それを各 DAQ コンポーネントへ送信しデータ収集システムの制御を行います。各コンポーネントのステータス情報等を取得しユーザや上位のフレームワークへ伝えます。また、XML 文書による DAQ システムのコンフィグレーション機能、XML/HTTP プロトコルを使用したシステム・インターフェイス機能を持っています。

第 2 節で DAQ ミドルウェアのアーキテクチャの概要、第 3 節で DAQ コンポーネントの仕様について説明します。第 4 節で DAQ オペレータの仕様を説明します。DAQ ミドルウェアの概要のみを知りたい方は、「DAQ ミドルウェア概要 [4]」をご覧ください。

2　DAQ ミドルウェアのアーキテクチャ

DAQ ミドルウェアのアーキテクチャについて下記の項目の説明をします。

- ソフトウェア・コンポーネント・モデル (2.1)
- 状態遷移モデル (2.2)
- ランコントロール・モデル (2.3)
- データ転送機能 (2.4)
- コマンド・ステータス通信機能 (2.5)
- システム・コンフィグレーション機能 (2.6)
- システム・インターフェイス機能 (2.7)
- 装置パラメータ設定機能 (2.8)
- リモートブート機能 (2.9)

2.1 ソフトウェア・コンポーネント・モデル

DAQ コンポーネントは、DAQ ミドルウェアにおけるソフトウェアの基本単位です。色々な機能の DAQ コンポーネントを組み合わせることで、柔軟な DAQ システムを構築できます。ソフトウェア・コンポーネント指向のフレームワークを用いることで、次のことが期待できます。

- 柔軟な DAQ システムの構築の実現
- ソフトウェア開発効率の向上
- ソフトウェア再利用性の向上
- ソフトウェアメンテナンス容易性の向上

前述の RT ミドルウェアにおけるソフトウェアの基本単位は RT コンポーネントです。ロボットシステムを構築するためには、センサ等のデバイスを組み合わせて実現しますが、その機能要素をソフトウェア・コンポーネントで実現し、複数組み合わせることによりシステムを構築することが可能です。RT コンポーネントのアーキテクチャを図 2 に示します。DAQ コンポーネントを説明する上で重要な RT コンポーネントの要素は以下のものです。

- 状態（ステート）
- 任意の数のデータ入力ポート (InPort)
- 任意の数のデータ出力ポート (OutPort)
- ユーザが定義可能なサービスポート

その他の機能や要素については、RT ミドルウェアのページ等 [1, 5] を参照してください。RT コンポーネントの状態遷移モデルについては後述します。InPort, OutPort は RT コンポーネント間を流れるデータストリームの入出力ポートです。Publisher/Subscriber モデルに基づきコンポーネント間のデータの送受信を抽象化しています。DAQ コンポーネントでも同様に、InPort, OutPort をデータの送受信に使用しています。サービスポートはユーザが任意のサービスインターフェースを定義できるポートです。「サービスプロバイダ」はサービスを提供するためのインターフェースで「サービスコンシューマ」はサービスを利用するためのインターフェースです。DAQ コンポーネントでは、このサービスポートを利用して DAQ オペレータと DAQ コンポーネント間のコマンド/ステータスの送受信を実装しています。

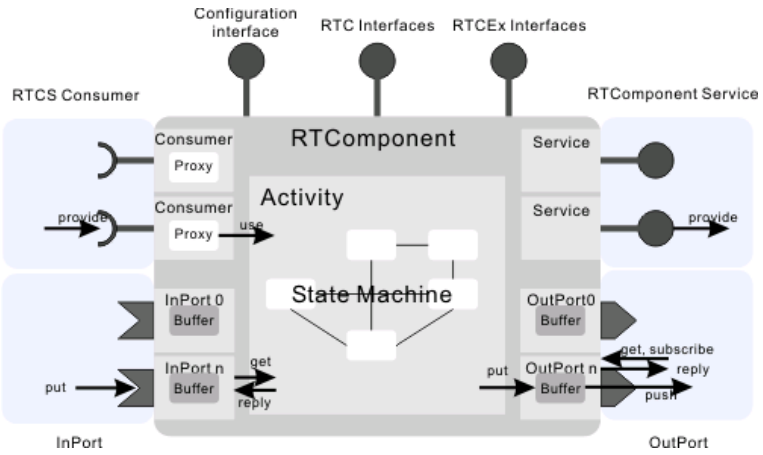

図 2: RT コンポーネントのアーキテクチャ

2.2 状態遷移モデル

図 3 に RT コンポーネントのステートチャートを示します。RT コンポーネントは”Active”, ”Inactive”, ”Error”という状態間を遷移します。図 4 に DAQ コンポーネントのステートチャートを示します。我々の考える一般的な DAQ システムの 4 つの状態、すなわち、システムが起動直後の状態 ”Loaded”, システムの構成に必要なパラメータを設定後の状態”Configured”, ランが実行中の状態 ”Running”, ポーズ状態 ”Paused” を RT コンポーネントの状態へ素直にマッピングすることはできないため、RT コンポーネントの ”Active” 状態のサブ・ステートとして実装しました。これにより、RT コンポーネントの状態遷移モデルを変更せずに DAQ に必要な状態を拡張することができました。図 5 に DAQ コンポーネントのコマンドによる状態遷移モデルを示します。

- DAQ コンポーネントが起動すると”Loaded”状態となります。
- ”Loaded”状態では”Configure”コマンドにより DAQ コンポーネントのパラメータの設定等を行い”Configured”状態になります。
- ”Configured”状態では”Start”コマンドにより ”Running”状態へ遷移します。
- ”Running”状態では ”Pause”コマンドにより ”Paused”状態へ遷移します。
- ”Paused”状態では ”Resume”コマンドで”Running”状態へ遷移します。

2.3 ランコントロール・モデル

現在の DAQ ミドルウェアのランコントロール・モデルは、1 階層のツリー構造です。1 つの DAQ オペレータ（コントローラ）で、すべての DAQ コンポーネントを制御します。今後、より大規模なシステムに対応できるように、多重階層化することも検討しています。前述のコンフィグレーション・ファイルに DAQ コンポーネントの起動の順番が記述します。現在の DAQ コンポーネント起動の順序は、それらを流れるデータストリームに対して下流から起動を開始し、停止は上流から行うように実装されています。コマンドは DAQ オペレータが各コンポーネントに対しコンフィグレーションファイルで指定された順番に送信するため、コンポーネントの数が多くなるとランの開始・停止にかかるオーバヘッドが大きくなります。

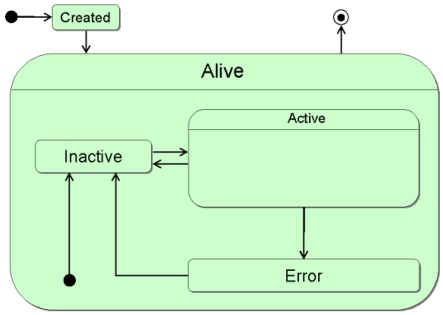

図 3: RT コンポーネントのステートチャート

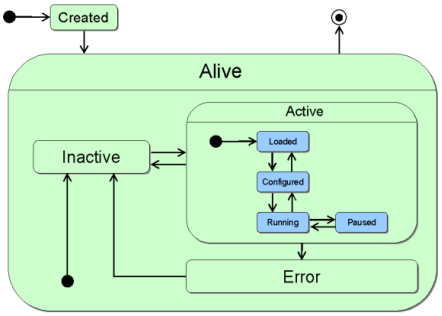

図 4: DAQ コンポーネントのステートチャート

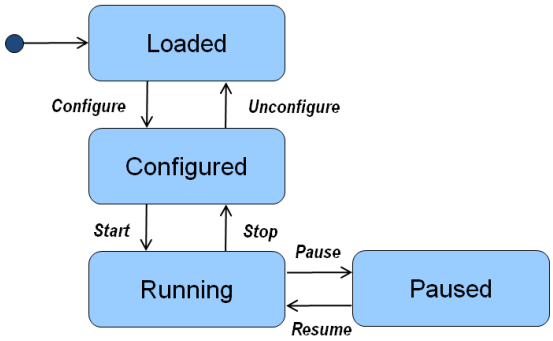

図 5: DAQ コンポーネントのコマンド（イタリック表記）と状態遷移

2.4 データ転送機能

DAQ コンポーネント間のデータ転送は、RT コンポーネントの InPort, OutPort を使用します。InPort, OutPort は、抽象化された RT コンポーネントの通信インターフェイスで、データを送受信するために作られたものです。DAQ コンポーネント間は、同一ホスト内でもネットワーク分散環境でも透過的にデータ転送を行うことができます。データ転送には **OpenRTM-aist-1.0.0** の実装に使われている OmniORB が使用されます。OmniORB はフリーの CORBA(Common Object Request Broker Architecture) の一つです。CORBA は様々な OS や言語で書かれたネットワーク上のソフトウェアの相互利用を可能にするフレームワークです。

DAQ コンポーネント間を流れるデータには 8 バイトのヘッダとフッタがついており、データの妥当性の検証に使用します。ヘッダには 4 バイトのマジック番号（0xe7e7）とデータのバイトサイズを 4 バイトで格納します。実際に取得したデータのバイトサイズとヘッダに格納されているバイトサイズを比較することでデータの妥当性を検証できます。ヘッダには予備として残り 2 バイトありますが、これはユーザが使用することができます。フッターには 4 バイトのマジック番号（0xcccc）とシーケンス番号と呼ばれる 4 バイトの数値があります。シーケンス番号は、コンポーネントの転送処理に応じて増加する値です。この数値によりデータに欠損がなかったか検証できます。下記にヘッダとフッタのフォーマットを示します。

表 1: ヘッダデータ

Header Magic (0xe7)	Header Magic (0xe7)	Reserved	Reserved	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)	Reserved	Reserved	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

2.5 コマンド・ステータス通信

DAQ コンポーネントとそのコントローラである DAQ オペレータの通信はサービスポートを使用しています。前述のように RT ミドルウェアのサービスポートは InPort や OutPort とは異なり、ユーザが任意のサービスインターフェースを定義できるポートです。サービスポートは「サービスプロバイダ」、「サービスコンシューマ」モデルにより CORBA で実装されています。DAQ コンポーネントは「サービスプロバイダ」で、DAQ オペレータは「サービスコンシューマ」に対応します。DAQ コンポーネントのコントローラである DAQ オペレータからの要求（コマンド）により DAQ コンポーネントは、その状態を遷移させます。サービスのインターフェイスを定義するために CORBA の IDL(Interface Definition Language) を使用します。DAQ ミドルウェアでは、DAQService.idl というファイルにインターフェイスの定義があります。

2.6 システム・コンフィグレーション

DAQ ミドルウェアでは、複数の DAQ コンポーネントを自由に接続してデータ収集システムを構築できます。それらの DAQ コンポーネントはローカル計算機あるいはリモート計算機上で動いていても同様です。このように使用する DAQ コンポーネントの構成、それらの接続をシステムコンフィグレーションと呼んでいます。その情報はコンフィグレーション・ファイル（config.xml）という XML 文書により記述します。XML は現在、広く普及している技術で、構造化された文書やデータを異なる情報システム間で共有できます。”Configure”コマンドによるシステムコンフィグレーション時にコンフィグレーションファイルを読み込みシステム構成を行います。コンフィギュレーションファイルの内容を変更することで、使用する DAQ コンポーネントを選択しコンポーネント間の接続を変更して柔軟に DAQ システムの変更が可能です。このようにコンフィグレーションファイルは DAQ システムを記述する広義のデータベースとしての意味を持ちます。XML 文書の文書構造をスキーマ言語により定義することができ、XML 文書の構造の妥当性の検証に使用します。コンフィグレーション・ファイルのスキーマ (config.xsd) は W3C の XML Schema で記述されています。コンフィグレーション・ファイルには DAQ コンポーネントを起動する計算機の IP アドレス、DAQ コンポーネントの名前、使用するデータ入出力ポートの種類と名前、起動順番等を記述します。DAQ オペレータは、コンフィグレーション・ファイルを解釈して必要な DAQ コンポーネントをネットワーク上から探して、コマンド・ステータス通信のため自身のサービスポートと DAQ コンポーネントのサービスポートを接続します。また、config.xml の記述にしたがい DAQ コンポーネント間のデータ入力ポートと対応するデータ出力ポートを接続してデータストリームの経路を確立します。詳細は 4.3 で説明します。

2.7　システム・インターフェイス

システム・インターフェイスとは、DAQ ミドルウェアと外部システムを接続する際に用いるインターフェイスです。外部システムとのインターフェイスは、より一般的な通信プロトコルを用いることが望ましいという理由で XML データを HTTP で転送する方式を採用しています。このプロトコルによる通信であれば、どのような言語、アプリケーションであってもシステムインターフェイスを介してランの制御が可能です。例えば Web ブラウザから DAQ オペレータへコマンドを送信してランコントロールをすることも可能です。J-PARC MLF 中性子実験においては上位のシステムである「IROHA」とこのプロトコルを用いて DAQ ミドルウェアによるデータ収集システム・サブシステムと通信を行っています。

システムインターフェイスの詳細は *DAQ* オペレータの説明の *4.6* で説明します。

2.8　装置および解析パラメータ設定機能

前述のコンフィグレーション・ファイルとは別に装置パラメータやオンライン解析パラメータ情報の広義のデータベースとして XML で書かれたコンディション・ファイルがあります。前述のコンフィグレーション・ファイルには実験のラン毎には変化しない DAQ システムの構成等の情報を記述し、コンディション・ファイルにはラン毎に変化しうる実験装置のパラメータやオンライン解析用のパラメータ等を記述します。コンディション・ファイルの詳細は 3.9 で説明します。コンフィグレーション・ファイルは DAQ オペレータが読み込んで情報を取得しますが、コンディション・ファイルは各コンポーネントが各自それを読み込んで必要な情報を取得します。現在は、各コンポーネントでの XML 構文解析処理の負荷を軽減するため、XML に比べて処理が軽量な JSON(JavaScript Object Notation) 形式に変換したファイルを各コンポーネントが読み込んで処理を行う方式をとっています。図 6 にコンフィグレーション・ファイルとコンディション・ファイルについて、それぞれの役割を示します。

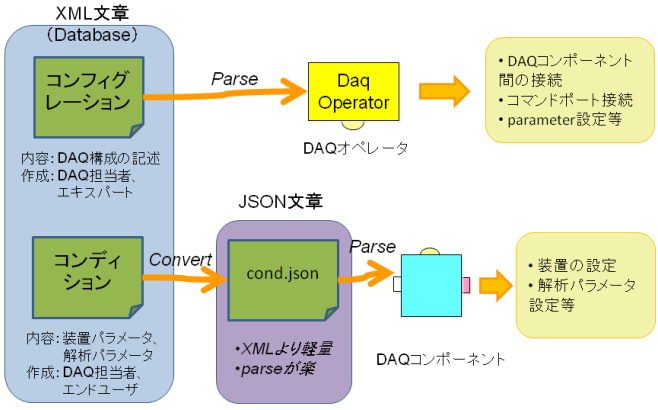

図 6: コンフィグレーション・ファイルとコンディション・ファイル

2.9　リモートブート機能

DAQ ミドルウェアのリモートブート機能とは、ネットワーク上の計算機 (CPU DAQ) にある DAQ コンポーネントをローカル計算機 (CPU UI) からのコマンドにより起動させる機能です。現在の実装では Linux

xinetd のサービスのひとつとしてポート 50000 番に対して動作を行います。J-PARC MLF 中性子におけるリモートブートの利用例を図 7 に示します。DAQ システム起動スクリプト run.py を起動します。各コンポーネントの起動メカニズムを図 7 をもとに解説します。

1. ユーザーがローカル計算機のコマンドプロンプトから `run.py` を起動します。`run.py` はまず、各コンポーネントがネームサーバーと通信するのに必要となる情報が書かれたファイル `rtc.conf` を作成し、各リモート計算機にネットワークを通じて `rtc.conf` を送ります。リモート計算機側ではこのファイルを受信するために xinetd から起動される受信サーバー `bootComps.py` を使います。受信したファイルは `/tmp/rtc.conf` に保存されます。なお `rtc.conf` の内容は各リモート計算機の構成にマッチしている必要があります。

2. `run.py` は続いてコンポーネント起動用スクリプト `run-comps.sh` ファイルを各リモート計算機に送ります。リモート計算機側では `rtc.conf` ファイルと同様に xinetd から起動された受信サーバー `bootComps.py` を使って `run-comps.sh` を受信します。受信したファイルは `/tmp/run-comps.sh` として保存されます。

3. xinetd から起動された `bootComps.py` は、`run-comps.sh` を受信後 `/tmp/run-comps.sh` を `system()` 関数で実行します。これで各コンポーネントが起動します。

4. 起動したコンポーネントは `/tmp/rtc.conf` を参照し、そこに書かれた情報をもとに Naming service へ自身を登録します。

5. 全てのリモート計算機へコンポーネントの起動をリクエストした後、`run.py` から DaqOperator が起動されます。

6. 起動した DaqOperator はローカルにある `config.xml` をパーズし、必要なコンポーネントを Naming service へ問い合わせ、コンポーネントを検索し、各コンポーネント間を接続します。Naming service に問い合わせて目的のコンポーネントが見つからない場合は、0.5 秒スリープして 20 回のリトライを行ないます。20 回のリトライでコンポーネントが見つからない場合は、エラーとなり、コンポーネントの起動に失敗します。すべてのコンポーネントが見つかった場合は、各コンポーネント間のデータポートが接続され LOADED 状態になります。

3 DAQ コンポーネントの仕様

3.1 関連ファイル

DAQ コンポーネントを開発する際に必要となる主なファイルを下記に示します。これらのファイルは **DAQ-Middleware 1.1.0** をインストールすると `/usr/include/daqmw` の下に置かれます。

- DaqComponentBase.h
- DaqComponentException.h
- FatalType.h
- idl/DAQService.idl

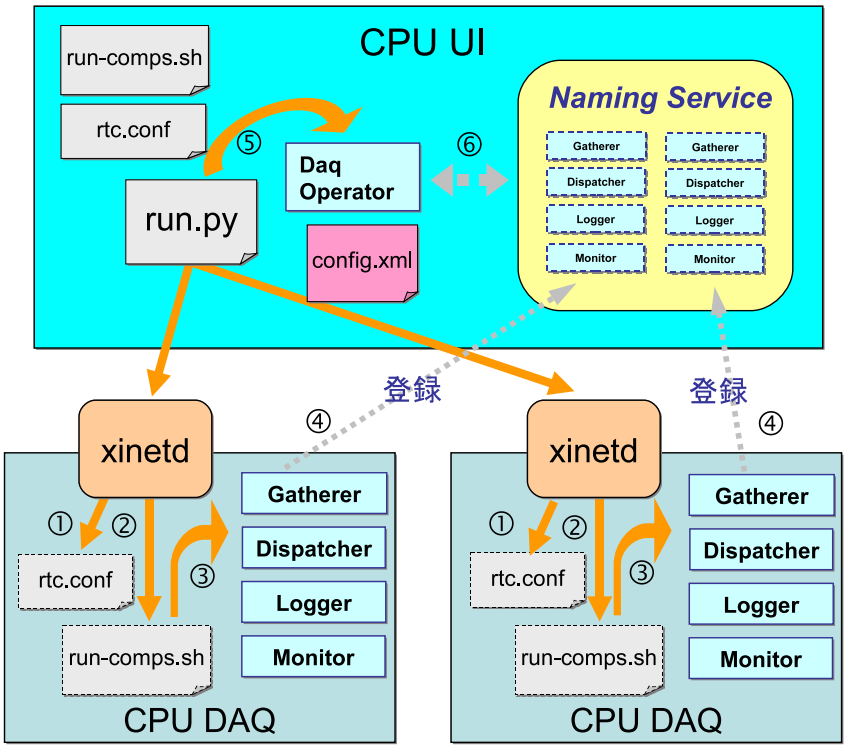

図 7: DAQ コンポーネントのリモート起動メカニズム。

DaqComponentBase.h には、DaqComponentBase クラスの定義と実装があります。DaqComponentException.h は、DaqComponentBase クラスの例外の定義があります。DAQ コンポーネントで致命的 (Fatal) なエラーが発生した際に例外が起きます。FatalType.h は定義済みの Fatal エラーが書かれています。verb—idl/DAQService.idl—には DAQ オペレータとのコマンド受信/ステータス送信の通信に使用するサービスがインターフェイス定義言語で書かれています。

3.2 DaqComponentBase クラス

DaqComponentBase クラスは、各 DAQ コンポーネント共通の機能の実装のために導入されました。DAQ ミドルウェアではユーザが独自の DAQ コンポーネントを開発してそれらを接続し、DAQ システムを構築します。新たな DAQ コンポーネントを開発する際は、この DaqComponentBase クラスを継承して新たなクラスを作ります。この継承により DAQ コンポーネントとして必要とされる機能は、実装されることになります。しかし継承しただけでは、何もしない（ロジックが空の）DAQ コンポーネントができるので、開発者は、各状態での動作の実装を行うことで必要な機能を実現することができます。詳細は 3.4 で説明します。また具体的な DAQ コンポーネントの開発手順は、[6] をご覧ください。DAQ コンポーネントを開発するユーザは、各状態および状態遷移の実装のみ行なえばよいので、開発効率、ソースコードのメンテナンスビリティの向上が図られます。

DaqComponentBase クラスは、図 8 のような継承関係を持っています。データフロー型の RT コンポーネントは、RTC::DataFlowComponentBase を継承して実装します。DAQ コンポーネントはデータフロー型 RT コンポーネントから拡張して作られているので、図 8 のような継承関係があります。DAQ コンポーネントを開発する際は、DAQMW::DaqComponentBase を継承して実装します。

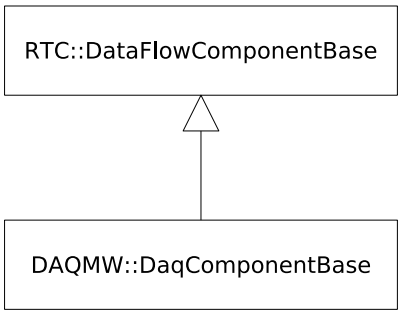

図 8: DAQ コンポーネントのクラス図

DaqComponentBase クラスで実現している機能は次のものです。

- DAQ のための状態遷移機能
- コマンド受信/ステータス送信機能
- Fatal エラーの際のステータス送信機能

DaqComponentBase クラスを継承して使用できるメンバ関数について説明します。その一覧を付録 B に示します。

3.2.1 コンポーネント初期化

コンポーネントのコンストラクタで使用する関数として、コマンドポートの初期化を行なう init_command_port() , 状態遷移テーブルの初期化を行なう init_state_table(), コンポーネントの名前を設定する set_comp_name() があります。

int init_command_port()	コマンドポート初期化
void init_state_table()	状態遷移テーブル初期化
int set_comp_name(char* name)	コンポーネント名の設定

3.2.2 シーケンス番号

DAQ コンポーネントはシーケンス番号と呼ばれる値を持っています。これは、データの入出力を行なうコンポーネントが期待される処理を行った際に 1 づつ増やされる値です。例えば、検出器からデータを取得するリーダコンポーネントは、データを取得しそのデータにヘッダとフッタをつけ、後段のコンポーネントに送信が成功した場合に 1 つ増加させます。コンポーネントの処理が完了したら inc_sequence_num() を呼んでシーケンス番号をインクリメントします。またシーケンス番号のリセットや取得するためには下記の関数を使います。データストリームの経路上のコンポーネントは、このデータ中のシーケンス番号と自身のシーケンス番号を比べることでデータの欠落がないかチェックが可能です。例えば、データを保存するコンポーネントではこのチェックは必須ですが、データのサンプリングによりオンラインモニタリングを行うコンポーネントでは必要ありません。

`int inc_sequence_num()`	シーケンス番号を 1 つ増加させる
`int reset_sequence_num()`	シーケンス番号を 0 にする
`unsigned long long get_sequence_num()`	シーケンス番号を取得する

3.2.3 転送データサイズ

データの入出力を行なう DAQ コンポーネントは、これまで自身が受信（または送信）したデータの総バイト数を持っています。データストリームのスタートポイントとなるコンポーネントからエンドポイントとなるコンポーネントまで、その値は等しくなることが要請されます（その間にデータをサンプリングしたりフィルタリングするコンポーネントが存在しない場合)。例えば、ラン終了後の検出器からデータを読み出すコンポーネントとデータの保存を行うコンポーネントの転送データサイズは等しいことが要請されます。DAQ コンポーネントの実装で転送データサイズのインクリメント、リセット、取得を行なうためには下記の関数を使います。

`int inc_total_data_size(unsigned int byteSize)`	指定したバイト数を総データバイト数に加える
`int reset_total_data_size()`	総バイト数を 0 にする
`unsigned long long get_total_data_size()`	総バイト数を取得する

3.2.4 データヘッダ、フッタ関連

2.4 で説明したヘッダ情報、フッタ情報をセットする関数として次の 2 つが用意されています。データバイト数をヘッダに格納する `set_header()`、シーケンス番号をフッタにセットする `set_footer()` です。データバイト数やシーケンス番号は、データを受信した際にデータの妥当性の検証に使用します。

`int set_header(unsigned char* header, unsigned int data_byte_size)`	ヘッダにデータバイト数を格納
`int set_footer(unsigned char* footer)`	フッタにシーケンス番号を格納

ヘッダ情報、フッタ情報をチェックするための関数として次の 3 つがあります。`check_header()`, `check_footer()` ではチェックした結果が bool 値で返ります。現在の仕様では、その値が False だった場合は、`fatal_error_report()` を呼んで Fatal エラーを報告し、自身はアイドル状態になり次のコマンドを待ちます。`check_header_footer()` では、その中で `check_header()`, `check_footer()` を呼んでおり、ヘッダまたはフッタに問題がある場合は、その関数中で `fatal_error_report()` を呼んでいます。 **DAQ-Middleware 1.1.0** では、これまで `set_footer()`, `check_footer()` の引数の `seq_num`(シーケンス番号) が省略されましたので注意してください。

`bool check_header(unsigned char* header, unsigned received_byte)`	ヘッダのチェックを行なう
`bool check_footer(unsigned char* footer)`	フッタのチェックを行なう
`bool check_header_footer(const RTC::TimedOctetSeq& in_data, unsigned int block_byte_size)`	
上記の 2 つを行なう	

`get_event_size()` はコンポーネント間で受信したデータのバイト数からヘッダとフッタを除いた正味のデータバイト数を返します。

`unsigned int get_event_size(unsigned int block_byte_size)`	正味のデータ数を取得する

3.2.5 状態遷移関連

DaqComponentBase 中では、ストップコマンドを受信すると RUNNING 状態から CONFIGURED 状態への遷移（状態遷移に関しては 3.4 で説明）がロックされます。コンポーネントが正しく CONFIGURED 状態に遷移するためには、現在処理中の動作を完了する必要があるからです。つまりストップコマンドを受信して直ちに状態遷移を行うのではなく、必要な処理を行った後に遷移を行うための仕掛けです。処理の完了の定義はコンポーネントによって違うので、そのロックを解除するのはコンポーネント自身です。具体的には、daq_run() 中の中断可能な処理のポイントでストップコマンドの有無を確認する check_trans_lock() を呼んで、その返り値が真の場合は、set_trans_unlock() を呼び、次の状態に遷移可能であることを知らせます。

```
bool check_trans_lock()  ストップコマンドが発行されているかチェックする
void set_trans_unlock()  RUNNING 状態から CONFIGURED 状態へ遷移を行なう
```

3.2.6 致命的（Fatal）エラー処理関連

DAQ コンポーネントはユーザがその目的に応じて自由に開発できます。したがって DAQ コンポーネントでは、種々のエラーが起こる可能性があります。DAQ ミドルウェアでは Fatal エラーは、「エラーが起きた場合そのコンポーネント自身で解決できないもの」と定義しています。その場合、下記の関数を呼んで DAQ オペレータに報告し、自身はアイドル状態になります。現在の仕様では、Fatal エラーが起きた場合は、システム全体をリセットする必要があります。複数のコンポーネントを使い、複数のデータストリームが存在するようなシステムでは、問題の起きたコンポーネントのみをリセットするだけでは、システム全体としての整合性を保証するのが難しいからです。ラン中に起きた Fatal エラーをリセットするためにはユーザまたは上位のフレームワークから Stop コマンドを発行します。システム全体は、RUNNING 状態から CONFIGURED 状態へ遷移しコンポーネントの Fatal エラーはリセットされます。ストップコマンドにより Fatal エラーがリセットされない場合は、全コンポーネントの再立ち上げが必要です（起動スクリプト run.py を再実行する。その際コンポーネントの再立ち上げが行われる）システムの Configure 時（Configure コマンド発行時）に起きた Fatal エラーは、Unconfigure コマンドによりシステムが LOADED 状態へ遷移する際にリセットされます。Unconfigure コマンドによりリセットされない場合は、前述の全コンポーネントの再立ち上げが必要です。DAQ ミドルウェアでは、Fatal エラーのタイプとして DAQ ミドルウェアで定義しているものと、ユーザが定義するものの 2 つに分類しています。詳細は 3.8 で説明します。DAQ ミドルウェアで定義済のエラーは enum でその種類を指定します。ユーザによる定義のものは、enum として USER_DEFINED_ERROR1,...,USER_DEFINED_ERROR20 から他のエラーと重複しないように指定して、エラーの詳細を文字列で指定します。これは、上位のフレームワークでユーザ定義の enum に対応した処理を行なう場合に有効です。

3.2.7 転送ステータス取得

DAQ-Middleware 1.1.0 では、下記の関数でデータポートのステータスを取得できます。

```
BufferStatus check_outPort_status(RTC::OutPort<RTC::TimedOctetSeq> & myOutPort)
指定した OutPort の転送ステータスを取得する

BufferStatus check_inPort_status(RTC::InPort<RTC::TimedOctetSeq> & myInPort)
指定した InPort の転送ステータスを取得する
```

返り値の BufferStatus は、次のような enum です。

```
enum BufferStatus {BUF_FATAL = -1, BUF_SUCCESS, BUF_TIMEOUT, BUF_NODATA, BUF_NOBUF}
```

OutPort からの転送が正常に終了した場合 BUF_SUCCESS を返します。タイムアウトが発生した場合は BUF_TIMEOUT を返し、送信先ののバッファがフルの場合は、BUF_NOBUF を返します。それ以外のエラーは、BUF_FATAL を返します。現在の仕様では、BUF_TIMEOUT、BUF_NOBUF の場合はデータ転送を再試行し、BUF_FATAL の場合は Fatal エラーにします。InPort からの転送が正常に終了した場合 BUF_SUCCESS を返します。タイムアウトの場合 BUF_TIMEOUT を返し、相手のバッファが空の場合 BUF_NODATA を返します。それ以外のエラーは、BUF_FATAL を返します。現在の仕様では、BUF_TIMEOUT、BUF_NODATA の場合はデータ転送を再試行し、BUF_FATAL の場合は Fatal エラーにします。

下記の関数でデータポートの接続を確認できます。例えば CONFIGURED 状態から RUNNING 状態へ遷移する際に、daq_start() 中で呼んでデータポートの接続を確認し daq_run() でデータの転送を行ないます。

```
bool check_dataPort_connections(RTC::OutPort<RTC::TimedOctetSeq> & myOutPort)
OutPort の接続を確認

bool check_dataPort_connections(RTC::InPort<RTC::TimedOctetSeq> & myInPort)
InPort の接続を確認
```

DaqComponentBase クラスを継承して作る DAQ コンポーネントは、下記の状態遷移に関係する下記の仮想関数を実装する必要があります。どのように実装するかは 3.4 で説明します。

```
...
virtual int daq_dummy()       = 0;
virtual int daq_configure()   = 0;
virtual int daq_unconfigure() = 0;
virtual int daq_start()       = 0;
virtual int daq_run()         = 0;
virtual int daq_stop()        = 0;
virtual int daq_pause()       = 0;
virtual int daq_resume()      = 0;
...
```

3.3 DAQ コンポーネント開発の実際

DAQ コンポーネント開発に必要なファイルについて説明します。例えば、Skeleton という名前のコンポーネントを開発するためには、次のファイルが必要です。これは RT コンポーネントのファイル構成に由来するものです。

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

Skeleton.h は、Skeleton クラスのヘッダファイルです。Skeleton.cpp には Skeleton コンポーネントの各状態でのロジックを実装します。詳細は 3.4 で説明します。SkeletonComp.cpp は、Skeleton コンポーネントのメインプログラムです。下記にその一部分を示します。RTC::Manager は RT コンポーネントの情報管理を行うクラスです。

- manager->init(argc, argv) により config ファイルの読み込み、Naming Service の初期化等を行います (7 行目)。
- manager->setModuleInitProc(MyModuleInit) によりコンポーネントの初期化プロシージャが設定されます (11 行目)。
- manager->activateManager() によりコンポーネントの生成を行います (14 行目)。
- manager->runManager() でマネージャのメインループを実行します (21 行目)。このメインループ内では、CORBA ORB のイベントループ等が処理されます。デフォルトでは、このオペレーションはブロックします。

```
1  int main (int argc, char** argv)
2  {
3      RTC::Manager* manager;
4      manager = RTC::Manager::init(argc, argv);
5
6      // Initialize manager
7      manager->init(argc, argv);
8
9      // Set module initialization proceduer
10     // This procedure will be invoked in activateManager() function.
11     manager->setModuleInitProc(MyModuleInit);
12
13     // Activate manager and register to naming service
14     manager->activateManager();
15
16     // run the manager in blocking mode
17     // runManager(false) is the default.
18     manager->runManager();
19
20     // If you want to run the manager in non-blocking mode, do like this
21     // manager->runManager(true);
22
23     return 0;
24 }
```

3.4 状態および状態遷移の実装

DAQ コンポーネントの DaqComponentBase クラスには、状態遷移の際に呼ばれるメンバ関数、その状態中に繰り返し呼ばれるメンバ関数が定義されています。DAQ コンポーネント開発者は、それらの関数の中を実装して新しい機能の DAQ コンポーネントを作ります。

図 9 に状態遷移の際に呼ばれるメソッド、その状態中に繰り返し呼ばれるメソッドを示します。また状態遷移のトリガとなるコマンドを示します。コマンドは、DAQService.idl に次のように enum で定義されています。

```
enum DAQCommand
{
    CMD_CONFIGURE,
    CMD_START,
    CMD_STOP,
    CMD_UNCONFIGURE,
    CMD_PAUSE,
    CMD_RESUME,
    CMD_NOP
};
```

また、コマンドに対応する状態は、次のように enum で定義されています。

```
enum DAQLifeCycleState
{
    LOADED,
    CONFIGURED,
    RUNNING,
    PAUSED
};
```

1. DAQ コンポーネントは起動時は、”LOADED” 状態で待機状態
   現在の実装では、”LOADED” 状態中は、daq_dummy() が繰り返し呼ばれる。daq_dummy() は CPU を消費しないように sleep() 関数を呼んでアイドル状態を実現している。

2. ”LOADED” 状態中は”Configure”コマンドを受けて”CONFIGURED”状態へ遷移
   その際、daq_configure() が一度呼ばれる。コンポーネントに対するパラメータの設定を行う。”CONFIGURED”状態では、daq_dummy() が繰り返し呼ばれアイドル状態。

3. ”CONFIGURED”状態中は”Start”コマンドで”RUNNING”状態へ遷移
   その際、daq_start() が一度呼ばれる。コンポーネントが連続動作を行う前の初期化やパラメータの設定等の処理を実装する。”RUNNING”状態中は、daq_run() が繰り返し呼ばれる。daq_run() にコンポーネントの機能のロジックを実装する。

4. ”RUNNING”状態中は”Pause”コマンドで”PAUSED”状態へ遷移
   その際、daq_pause() が一度呼ばれる。コンポーネントが”PAUSED”状態へ遷移する前に行う処理を実装する。”PAUSED”状態では daq_dummy() が繰り返し呼ばれアイドル状態。

5. ”PAUSED”状態中は”Resume”コマンドで”RUNNING”状態へ遷移
   その際、daq_resume() が一度呼ばれる。コンポーネントが”RUNNING”状態へ遷移する前に行う処理を実装する。

6. ”RUNNING”状態中は ”Stop”コマンドで”CONFIGURED”状態へ遷移
   その際、daq_stop() が一度呼ばれる。コンポーネントが連続動作を終了するための処理等を実装する。

7. ”CONFIGURED”状態中は”Unconfigure”コマンドで”LOADED”状態へ遷移します。その際、daq_unconfigure() が一度呼ばれる。”CONFIGURED”状態中は daq_dummy() が繰り返し呼ばれアイドル状態。

3.4.1 LOADED 状態から CONFIGURED 状態への遷移の実装

LOADED 状態から CONFIGURED 状態へ遷移の際に、パラメータのリストが DAQ オペレータから送られてきます。各コンポーネントでは、daq_configure() 中でパラメータリストからパラメータの名前をキーにして検索を行いその値を取得します。

```
1 int Skeleton::daq_configure()
2 {
3     ...
4     ::NVList* paramList;
5     paramList = m_daq_service0.getCompParams();
6     parse_params(paramList);
7
8     return 0;
9 }
```

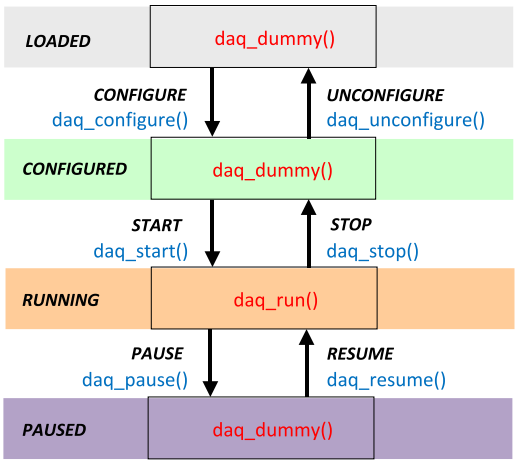

図 9: DAQ コンポーネントのステートチャート

3.4.2 RUNNING 状態の動作の実装

開発する DAQ コンポーネントの機能（メインロジック）を `daq_run()` に実装します。例えば、次のような動作の実装を行います。

- J-PARC MLF で仕様している Gatherer コンポーネントは、リードアウト・モジュールからデータを取得してデータにヘッダとフッタを付けて後段のコンポーネントへ送信する
- Logger コンポーネントは受信したデータからヘッダとフッタを取り除いてファイルへ保存する
- Monitor コンポーネントは受信したデータをデコードして、興味のある物理量等を計算してヒストグラム化する

3.5 コマンドの受信

DAQ オペレータから送信されるコマンドの受信は、`DaqComponentBase::daq_do()` の中の `get_command()` で行っています。これは前述のように、サービスポート間のデータ転送が CORBA により実現されています。現在のコンポーネントの状態（ステート）と受信したコマンドにより、状態遷移を行います。

3.6 ステータスの送信

DAQ コンポーネントは各状態（例えば ”RUNNING”状態）では定期的（現在の仕様では 2 秒毎）に、下記の構造体のデータを自身のステータスとして更新します。具体的には、コンポーネントの名前、状態（ステート）、イベント数、コンポーネントのサブ・ステータスです。ある状態から別の状態へ遷移した場合または致命的なエラーが発生した場合は、ステータス情報は、そのタイミングで更新されます。

```
struct Status
{
    ComponentName comp_name;
    DAQLifeCycleState state;
    unsigned long long event_size;
    CompStatus comp_status;
};
```

3.7 データ送受信

DAQ コンポーネントは、任意の数のデータ入出力ポートを持つことができます。これは RT コンポーネントから受け継いだ機能です。ユーザが開発を行なう機会が多いのは、データストリームの始点となるソース (source) タイプコンポーネントと終点となるシンク (sink) タイプコンポーネントです。例えば検出器やリードアウトモジュールからデータを取得して、後段のコンポーネントに送信するのがソースタイプです。前段のコンポーネントからデータを受信して、ファイルに保存するコンポーネントやデータをデコードしてヒストグラム等にしてオンラインモニタを行なうコンポーネントはシンクタイプです。データの受信を行うポートは InPort, 送信を行うポートは OutPort です。 **DAQ-Middleware 1.1.0** のパッケージに含まれる例題コンポーネントで InPort をもつものは、Dispatcher, SampleLogger, SampleMonitor 等のコンポーネントです。いずれも InPort の数は 1 個ですが、上述のように複数の InPort を持つコンポーネントを作ることは可能です。OutPort をもつ例題コンポーネントは、SampleReader, Dispatcher 等のコンポーネントです。OutPort を複数持つことは可能で、Dispatcher は 2 つの OutPort を持っています。複数の InPort, OutPort を持つコンポーネントでは、一般的にそのメインロジックが複雑になる傾向があります。

3.7.1 データ受信

InPort を持つコンポーネントは OutPort をもつコンポーネントからデータを受信できます。InPort には関連付けられたリングバッファがあり、OutPort から送信されたデータはこのバッファに書き込まれます。下記の `read()` によりタイムアウト付きのブロックモードでデータを読み込みます。正常にデータが読み込まれた場合は、返り値は `true` になります。false の場合は、`check_inPort_status()` により転送ステータスを調べます。`false` の場合、`check_inPort_status()` は、`BUF_TIMEOUT` または `BUF_FATAL` を返します。タイムアウトの場合はリトライを、Fatal の場合は、`fatal_error_report()` により報告します。

```
bool ret = m_InPort.read()
```

3.7.2 データ送信

OutPort を持つコンポーネントは InPort をもつコンポーネントへデータを送信できます。

下記の `write()` によりタイムアウト付きのブロックモードでデータを書き込みます。正常にデータが書き込まれた場合は、返り値は `true` になります。false の場合は、`check_outPort_status()` により転送ステータスを調べます。`false` の場合、`check_outPort_status()` は、`BUF_TIMEOUT` または `BUF_FATAL` を返します。タイムアウトの場合はリトライを、Fatal の場合は、`fatal_error_report()` により報告します。

```
bool ret = m_OutPort.write()
```

3.8 致命的エラー報告の送信

前述のように DAQ ミドルウェアでは、Fatal エラーのタイプとして DAQ ミドルウェアで定義しているものとユーザが定義するものの 2 つに分類しています。DAQ ミドルウェアで定義済のエラーは対応する FatalType::Enum で指定します。例えば受信したデータのヘッダの値に異常があった場合は HEADER_DATA_MISMATCH を指定します。ユーザによる定義のものは、USER_DEFINED_ERROR1,...,USER_DEFINED_ERROR20 から他のエラーと重複しないように指定して、エラーの詳細を文字列で指定します。これは、上位のフレームワークでユーザ定義の enum に対応した処理を行なう場合に有効です。その際、fatal_error_report() という関数を使用して DAQ オペレータへ Fatal エラーを報告します。

```
void fatal_error_report(FatalType::Enum type, int code = -1)
void fatal_error_report(FatalType::Enum type, const char* desc, int code = -1)
```

DAQ コンポーネント自身は、Fatal エラー報告後、アイドル状態となり次のコマンドを待ちます。これは各 DAQ コンポーネントで復旧不可能なエラーが発生した場合に、その情報を DAQ オペレータからユーザまたは上位システムに伝え、エラーの対応を行ってもらうためです。

3.2.6 の繰り返しになりますが、ラン中に起きた Fatal エラーをリセットするためにはユーザ（人）または上位のフレームワークから Stop コマンドを発行します。システム全体は、RUNNING 状態から CONFIGURED 状態へ遷移しコンポーネントの Fatal エラーはリセットされます。ストップコマンドにより Fatal エラーがリセットされない場合は、全コンポーネントの再立ち上げが必要です（起動スクリプト run.py を再実行する。その際コンポーネントの再立ち上げが行われる）システムの Configure 時（Configure コマンド発行時）に起きた Fatal エラーは、Unconfigure コマンドによりシステムが LOADED 状態へ遷移する際にリセットされます。Unconfigure コマンドによりリセットされない場合は、前述の全コンポーネントの再立ち上げが必要です。

様々な目的の DAQ コンポーネントが開発される可能性があるため、どのような状態を Fatal エラーにするかはコンポーネント開発者が自身で決めます。実装例としては下記のように、OutPort からデータを転送した後、そのステータスをチェックしてエラーの場合は、 fatal_error_report() を呼びます。

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

Fatal エラーのタイプは、FatalType.h で enum により下記のように定義してあります。

```
namespace DAQMW
{
    namespace FatalType
    {
        enum Enum
        {
            ///DAQ-Middleware defined fatal error
            ///use following function
            ///fatal_error_report(FatalTypes types, int code)
            /// e.g. fatal_error_report(HEADER_DATA_MISMATCH, -1)

            ///header, footer error
            HEADER_DATA_MISMATCH,
            FOOTER_DATA_MISMATCH,
            SEQUENCE_NUM_MISMATCH,

            ///configuration file
            CANNOT_OPEN_CONFIGFILE,
            CONFIGFILE_PARSE_ERROR,
            NO_CONFIG_PARAMS,
```

```
            ///condition file
            CANNOT_OPEN_COND_FILE,
            COND_FILE_PARSE_ERROR,

            ///command/status path error
            CANNOT_CONNECT_COMMANDPATH,
            COMMANDPATH_DISCONNECTED,

            ///data path error
            CANNOT_CONNECT_DATAPATH,
            DATAPATH_DISCONNECTED,

            ///InPort/OutPort error
            INPORT_ERROR,
            OUTPORT_ERROR,

            ///wrong parameters, such as command line options, etc.
            BAD_PARAMETER,

            ///readout module-related error
            CANNOT_CONNECT_DATA_SRC,
            TOO_MANY_DATA_FROM_DATA_SRC,
            READOUT_ERROR,

            ///file I/O error
            BAD_DIR,
            CANNOT_MAKE_DIR,
            CANNOT_OPEN_FILE,
            CANNOT_WRITE_DATA,

            ///user defined fatal error (user defined error1 - error20)
            ///users can choose a below error and its description by string.
            ///fatal_error_report(FatalTypes types, std::string desc, int code)
            /// e.g.
            /// fatal_error_report(USER_DEFINED_ERROR1,
            ///                                  "My fatal error detail", -1)
            USER_DEFINED_ERROR1,
            USER_DEFINED_ERROR2,
            USER_DEFINED_ERROR3,
            USER_DEFINED_ERROR4,
            USER_DEFINED_ERROR5,
            USER_DEFINED_ERROR6,
            USER_DEFINED_ERROR7,
            USER_DEFINED_ERROR8,
            USER_DEFINED_ERROR9,
            USER_DEFINED_ERROR10,
            USER_DEFINED_ERROR11,
            USER_DEFINED_ERROR12,
            USER_DEFINED_ERROR13,
            USER_DEFINED_ERROR14,
            USER_DEFINED_ERROR15,
            USER_DEFINED_ERROR16,
            USER_DEFINED_ERROR17,
            USER_DEFINED_ERROR18,
            USER_DEFINED_ERROR19,
            USER_DEFINED_ERROR20,

            ///unknown error
            UNKNOWN_FATAL_ERROR
        };
...
```

現在、ユーザが使用できる CompFatalTypes は、USER_ERROR1〜USER_ERROR20 の 20 個です。1 つの DAQ コンポーネント中で最大 20 個の Fatal エラーをユーザが定義できることになります。エラーの説明として文字列を指定できます。例えば、データ読み出し用のリーダコンポーネントが、読み出しボードに対してアクセスできなかった場合、下記のような実装になります。

```
if (user_defined_fatal2) {
    std::cerr << "### MyComponent: FATAL ERROR\n";
    fatal_error_report(USER_DEFINED_ERROR2, "COULD NOT ACCESS READOUT MODULE");
}
```

3.9 装置パラメータ設定機能

DAQ コンポーネントは、必要があれば装置パラメータやオンライン・モニタ用のパラメータをコンディション・ファイルと呼ばれる XML 文書から取得して、ランのスタート時に装置へ設定することが可能です。J-PARC MLF 中性子 PSD 検出器系の DAQ システムでは、NEUNET というリードアウト・モジュールに対して、PSD 検出器の信号のスレッショルド・レベルを設定する際に使用しています。この機能をどのように DAQ コンポーネントに実装するかは、[7] を参照してください。

4 DAQ オペレータの仕様

4.1 関連ファイル

DAQ オペレータ関連の主なファイルを下記に示します。これらのファイルは **DAQ-Middleware 1.1.0** をインストールすると `/usr/share/daqmw/DaqOperator` の下に置かれます。

- DaqOperator.h
- DaqOperator.cpp
- DaqOperatorComp.cpp
- ConfFileParser.h
- ConfFileParser.cpp
- CreateDom.h
- CreateDom.cpp
- ParameterServer.h
- Parameter.h
- callback.h

DAQ オペレータは、現在の実装では、RTC::DataFlowComponentBase クラスを継承しています。そのクラスの定義、実装が `DaqOperator.h,DaqOperator.cpp,DaqOperatorComp.cpp` に書かれています。コンフィグレーションファイルのパーズ関連の ConfFileParser クラスの定義と実装が `ConfFileParser.h,ConfFileParser.cpp` に書かれています。ユーザからのコマンドへの応答やコンポーネントのステータスを XML 形式にする CreateDom クラスの定義と実装が `CreateDom.h,CreateDom.cpp` にあります。HTTP 通信でユーザや外部システムとのインターフェイスの役割を担う ParameterServer クラスの定義と実装が `ParameterServer.h` にあります。コールバック関数操作関連の Parameter クラス関連の定義と実装が `Parameter.h` にあります。コールバック関数の定義は `callback.h` にあります。

4.2 DAQ オペレータの機能

DAQ オペレータは、ユーザや外部のフレームワークとのインターフェイスとなり DAQ コンポーネントの制御を行うソフトウェアです。DAQ オペレータは次のような機能を持っています。

- コンフィグレーションファイル（XML 文書）をパーズしてシステムの構成を行う機能 (4.3)
- DAQ コンポーネントへのコマンド送信/ステータス取得機能 (4.4,4.5)
- 各 DAQ コンポーネントへのパラメータ送信機能 (4.3)
- XML/HTTP プロトコルによる外部システムとのインターフェイス機能 (4.6)
- 標準入力からのコマンドによるランコントロール機能 (4.7)

上記の機能について次の 4.3 から 4.7 で説明します。

4.3 コンフィグレーション機能

2.6 で述べたように XML 文書による DAQ システムのコンフィグレーションが可能です。XML 文書には、次のような DAQ システムの情報が記述されています。

- DAQ オペレータの IP アドレス
- 使用する DAQ コンポーネントの IP アドレス
- 使用する DAQ コンポーネントのインスタンス名
- 個々の DAQ コンポーネントが使用する InPort, OutPort の名前
- DAQ コンポーネント間を流れるデータストリームの経路を決めるための（どの OutPort とどの InPort を接続するか）情報

システム・コンフィグレーションのシーケンスは次のようになっています。

1. DAQ コンポーネントは起動後、RT コンポーネントの機能を使い自分自身のインスタンス名を CORBA Naming Service へ登録します。
2. DAQ オペレータは起動後コンフィグレーション・ファイルを読み込み構文解析を行います。
3. DAQ オペレータは、コンフィグレーション・ファイルに記述されている DAQ コンポーネントを Naming Service へ問い合わせてそのオブジェクト・リファレンスを取得します。オブジェクト・リファレンスとは CORBA オブジェクトを識別するために用いられる参照です。
4. DAQ オペレータはそのオブジェクト・リファレンスを使って、DAQ コンポーネントのポート間の接続を行います。これによりコンポーネント間のデータストリームの経路が確立します。
5. DAQ オペレータは自身のサービスポートと各 DAQ コンポーネントのサービスポートの接続を行います。これによりコマンド送信、ステータス受信の経路が確立します。前述のように各コンポーネントが「サービスプロバイダ」で DAQ オペレータが「サービスコンシューマ」に対応します。

また DAQ オペレータは、”Configure”コマンドを各コンポーネントに送信する際に、コンフィグレーション・ファイルに書かれたパラメータを名前と値という組のリストにして各コンポーネントへ送信します。各コンポーネントでは送信されたリストから名前をキーにして値を取得します。 **DAQ-Middleware 1.1.0** の実装ではパラメータ名前はコンフィグレーションファイル内でユニークである必要があります（3.4.1）。

4.3.1 コンフィグレーションファイル

DAQ システムのコンフィグレーションに使用する XML 文書について説明します。はじめにコンフィグレーションファイルの構造を規定する XML スキーマファイル (config.xsd) について説明します。次にコンフィグレーションファイルの簡単な例を示します。

4.3.2 コンフィグレーションファイル用 XML スキーマ

XML のスキーマ言語としては数種類存在しますが、W3C の XML Schema を使用しています。config.xml のスキーマ config.xsd を付録 C に載せました。config.xml で使用可能なエレメントについて説明します。表 3 にエレメント名をまとめました。`configInfo` というルートエレメントの中に、`daqOperator` と `daqGroups` というエレメントがあります。`daqGroups` の中には複数の `daqGroup` が存在します。`daqGroup` は複数の `component` から構成されます。`component` には、`hostAdd`, `hostPort`, `instName`, `execPat`, `confFile`, `startOrd`, `inPorts`, `outPorts`, `param` エレメントがあります。`inPorts`, `outPorts`, `params` はそれぞれ、複数の `inPort`, `outPort`, `param` を持ちます。`param` エレメントは、各コンポーネントに特有なパラメータ等を記述する際に使用します。このパラメータは DAQ オペレータから configure コマンドと一緒に名前と値のリストとして各コンポーネントへ送られます。各コンポーネントは、その名前をキーにリストから値を取得し設定を行います。パラメータとその設定については後述します。

エレメント名	属 性	説 明
configInfo	なし	ルート・エレメント
daqOperator	なし	子エレメントとして 1 個の hostAddr をもつ
daqGroups	なし	子エレメントとして 1 個以上の daqGroup をもつ
daqGroup	gid	グループ ID を任意の文字列で指定する 例: `<daqGroup gid="group0">`
components	なし	子エレメントとして 1 個以上の daqComponent をもつ
component	cid	MyModuleInit() の `manager->createComponent("xxx")` で使用した文字列 ”xxx”に”0”を付加した文字列 例: `<component cid="Reader0">`
hostAddr	なし	コンポーネントを起動させるホストの IP Address 例: `<hostAddr>192.168.1.206</hostAddr>`
hostPort	なし	コンポーネントのリモート起動に使用する xinetd のポート番号 例: `<hostPort>50000</hostPort>` 50000 番を使用する
instName	なし	コンポーネントのインスタンス名。cid に”.rtc”を付加する 例: `<instName>Reader0.rtc</instName>`
execPath	なし	コンポーネントの実行形式ファイルの絶対パス
confFile	なし	コンポーネントの使用する rtc.conf ファイルの絶対パス
startOrd	なし	コンポーネントのスタートコマンド投入の際の順番
inPorts	なし	子エレメントとして 0 個以上の inPort をもつ
inPort	from	`registerInPort ("xxx",m_InPort)` で登録したの InPort 名”xxx”を指定 ”from”には接続する OutPort を指定する。形式は `cid : outPort` 例: `<inPort from="Reader0:reader_out">monitor_in</inPort>`
outPorts	なし	子エレメントとして 0 個以上の outPort をもつ
outPort	なし	`registerOutPort("xxx",m_OutPort)` で指定した outPort 名”xxx”を指定 例: `<outPort>reader_out</outPort>`
params	なし	子エレメントとして 0 個以上の param をもつ
param	pid	pid 属性にユニークなパラメータ名を指定する 例: データソースの IP アドレス`<param pid="srcAddr">192.168.0.80</param>`

表 3: コンフィグレーション・ファイルに使用するエレメント

4.3.3 コンフィグレーションの例

ここでは簡単なコンフィグレーションの例を示します。下記の XML で記述されていることを列挙します。

- ローカルホスト上の DAQ オペレータを使用する (4 行目)。
- DAQ コンポーネントは、Reader コンポーネント、Monitor コンポーネントを使用する (9 行目、26 行目)。
- Reader コンポーネントは、ローカルホストで起動する (10 行目)。
- Reader コンポーネントは、`reader_out` という名前の出力ポートを 1 つ持つ。入力ポートはない (19 行目、16,17 行目)。
- Monitor コンポーネントは、ローカルホストで起動する (27 行目)。
- Monitor コンポーネントは `monitor_in` という名前の入力ポートを 1 つ持つ。出力ポートはない (34 行目、36,37 行目)。
- `reader_out` と `monitor_in` は接続される (34 行目)。
- Reader コンポーネントの実行形式ファイルは、`/home/daq/MyDaq/Reader/ReaderComp` である (13 行目)。
- Monitor コンポーネントの実行形式ファイルは、`/home/daq/MyDaq/Monitor/MonitorComp` である (30 行目)。
- Reader コンポーネントは、pid が `srcAddr` でその値が `127.0.0.1` と pid が `srcPort` で値が `2222` という 2 組のパラメータを持っている (22,23 行目)。
- Monitor コンポーネントは、pid が `update_rate` で値が `100` という 1 組のパラメータを持っている (39 行目)。
- コンポーネントの起動順序は、Monitor, Reader（停止順序はその逆）である (15, 32 行目)。

```
1  <?xml version="1.0"?>
2  <configInfo>
3      <daqOperator>
4          <hostAddr>127.0.0.1</hostAddr>
5      </daqOperator>
6      <daqGroups>
7          <daqGroup gid="group0">
8              <components>
9                  <component cid="Reader0">
10                     <hostAddr>127.0.0.1</hostAddr>
11                     <hostPort>50000</hostPort>
12                     <instName>Reader0.rtc</instName>
13                     <execPath>/home/daq/MyDaq/Reader/ReaderComp</execPath>
14                     <confFile>/tmp/daqmw/rtc.conf</confFile>
15                     <startOrd>2</startOrd>
16                     <inPorts>
17                     </inPorts>
18                     <outPorts>
19                         <outPort>reader_out</outPort>
20                     </outPorts>
21                     <params>
22                         <param pid="srcAddr">127.0.0.1</param>
23                         <param pid="srcPort">2222</param>
24                     </params>
```

```
                    </component>
                    <component cid="Monitor0">
                        <hostAddr>127.0.0.1</hostAddr>
                        <hostPort>50000</hostPort>
                        <instName>Monitor0.rtc</instName>
                        <execPath>/home/daq/MyDaq/Monitor/MonitorComp</execPath>
                        <confFile>/tmp/daqmw/rtc.conf</confFile>
                        <startOrd>1</startOrd>
                        <inPorts>
                            <inPort from="Reader0:reader_out">monitor_in</inPort>
                        </inPorts>
                        <outPorts>
                        </outPorts>
                        <params>
                            <param pid="update_rate">100</param>
                        </params>
                    </component>
                </components>
            </daqGroup>
        </daqGroups>
    </configInfo>
```

4.4 コマンドの送信機能

DAQ オペレータは前述のサービスポートを利用して各コンポーネントへコマンドを送信します。各 DAQ コンポーネントは受信したコマンドと現在のステートに対応した状態遷移を行います。現在、DAQService.idl で定義されているコマンドを示します。

```
enum DAQCommand
{
    CMD_CONFIGURE,
    CMD_START,
    CMD_STOP,
    CMD_UNCONFIGURE,
    CMD_PAUSE,
    CMD_RESUME,
    CMD_NOP
};
```

コマンド送信のシーケンスは、次の通りです。

1. ユーザからコマンドを受信する
2. DAQ コンポーネントへコマンドを送信
3. DAQ コンポーネントからの acknowledge を待つ
4. 全コンポーネントに対し 2, 3 を行う

4.5 DAQ コンポーネント・ステータスの取得機能

DAQ コンポーネント・ステータスの取得は、前述のサービスポートを利用して各コンポーネントのステータスを取得します。各コンポーネントからのステータス受信動作を行う条件は、前述の DAQ オペレータの動作モードにより異なります。Web モードで動作中は、外部システムからのステータス取得コマンドにより DAQ オペレータが各 DAQ コンポーネントのステータスを取得して外部システムへ送信します。コンソール・モード動作中は、DAQ オペレータは定期的に（現在の実装では 2 秒毎）各 DAQ コンポーネントのステータス情報の取得を行い、標準出力へ表示を行います。

4.6 外部システムとのインターフェイス機能

ユーザが DAQ オペレータを介してランを制御するために 2 つの方法が用意されています。HTTP による XML で記述されたコマンドやコンポーネントのステータス等を Web サーバ経由で送受信する”Web モード”と標準入力からコマンドを入力する”コンソールモード”です。”Web モード”によるランコントロールが DAQ-Middleware 標準のモードです。”コンソールモード”は、デバッグやシステムのテスト等で使用します。

4.6.1 システムインターフェイスの概要

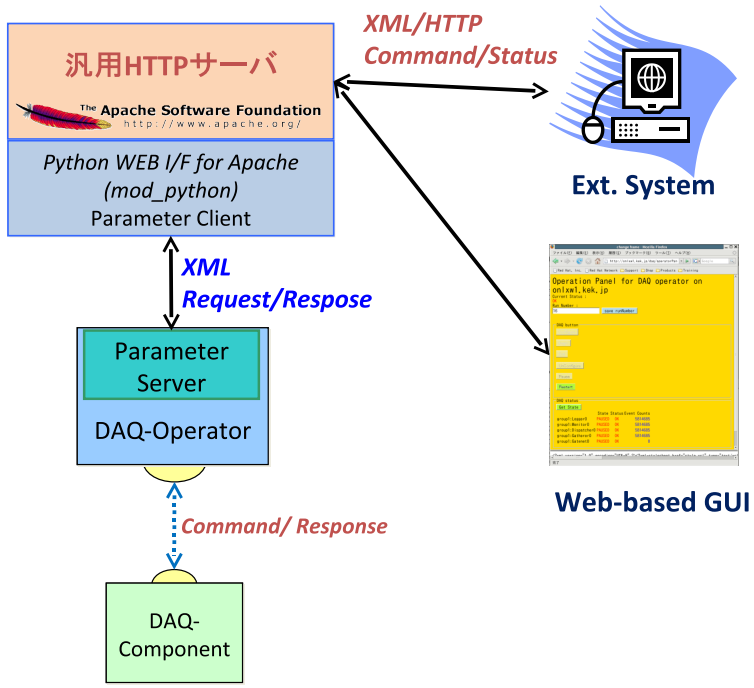

図 10: システム・インターフェイスの概念図

システム・インターフェイス機能の概念図を図 10 に示します。DAQ オペレータの”Web モード”では汎用 HTTP サーバ (Apache) と通信するための”Parameter Server”が起動します。”Parameter Server”は”Parameter Client”からの要求を待ち受けています。”Parameter Client”は Apache で Python を実行するためのモジュール `mod_python` により実装されています。付録 A に外部システムとの通信で使用するプロトコルの詳細を示します。

4.6.2 システムインターフェイスの実装

DAQ オペレータは起動の際のオプションで、”Web モード”と”コンソールモード”を指定します。それぞれのモードに対応して `DaqOperator::onExecute()` で下記のメソッドが呼ばれます。

- `DaqOperator::run_http_mode()`

- DaqOperator::run_console_mode()

ここでは、システムインターフェイス機能を担う”Web モード”の実装について説明します。”Web モード”の動作を起動から順番に説明します。”Web モード”で使用する”Parameter Server”クラスは、ParameterServer.h にその定義と実装があります。”Parameter Server”と”Parameter Client”は Socket による通信を行います。下記に DaqOperator::run_http_mode() と ParameterServer::Run() のソースコードの概略を示します。

```
RTC::ReturnCode_t DaqOperator::run_http_mode()
{
    if (g_server == NULL) {
        g_server = new DAQMW::ParameterServer(m_param_port);
        ...
        g_server->bind("put:Begin", &m_body, cb_command_start);
        g_server->bind("put:End", &m_body, cb_command_stop);
        ...
        g_server->bind("get:Log", &m_body, cb_command_log);
        ...
    }
    g_server->Run();
    run_data();

    return RTC::RTC_OK;
}
```

```
inline void* ParameterServer::Run() {
  try {
    m_server.accept ( m_newSock );

    int status;
    CallBackFunction callback;

    std::string command, com, value;
    int msgSiz=0;
    m_newSock.recvAll((unsigned int*)&msgSiz, 4);
    m_newSock.recvAll(command, msgSiz);

    status = extCmdVal(command, &com, &value);
    if (!status) {
      std::cerr << "Invalid command" << std::endl;
      std::string ng = "NG";
      unsigned siz = ng.size();
      m_newSock.sendAll(&siz, sizeof(siz));
      m_newSock.sendAll(ng);
    } else {
      if(com == "put") {
        m_msg = value;
        *(m_param.getValueP()) = value;
      }
      if(com == "get") {
        m_msg =  *(m_param.getValueP());
      }

      if ((callback = m_param.getCallBackFunc()) != (CallBackFunction)0) {
        (*(m_param.getCallBackFunc()))();
      }
      ...
      memcpy(&buf[1], m_msg.c_str(), size);
      m_newSock.sendAll(buf, length);
      delete [] buf;
      m_newSock.disconnect();
    }
  } catch ( SockException& e ) {
    std::cerr << "ParameterServer: Exception was caught:"
    << e.what();
  } catch (...) {
    std::cerr << "ParameterServer: Exception was caught"  << std::endl;
  }
  return 0;
}
```

1. DaqOperator::onExecute() から run_http_mode() が呼ばれる。
2. run_http_mode() では、”Parameter Server”を生成し、コマンドに対応したコールバック関数の登録を行う。その後 ParameterServer::Run() を呼ぶ。
3. ParameterServer::Run() では、”Parameter Client”からの接続を待つ。
4. ”Parameter Server”は”Parameter Client”との接続後にコマンドを受信してパーズを行う。
5. ”Parameter Server”はコマンドに対応したコールバック関数を呼ぶ
6. 各コンポーネントへのコマンド送信に問題がなければ、コールバック関数で生成された正常にコマンドを受け付けたという XML メッセージを”Parameter Client”へ送信する。
7. ”Parameter Client”との接続を切断する。

次に例としてスタートコマンドが発行された際の処理について説明します。

1. スタートコマンドに対応した関数 cb_command_start() が呼ばれる（cb は callback の意味）。callback 関数は、callback.h に記述されている。
2. 関数 cb_command_start() から DaqOperator::command_start() が呼ばれる。
3. DaqOperator::command_start() で DaqOperator::start_procedure() が呼ばれる。
4. DaqOperator::start_procedure() ではラン番号を各コンポーネントへ送信した後、スタートコマンドを送信する。
5. 例外が発生しなければ、コマンドを正常に受け付けたことを示す XML のメッセージを CreateDom::getOK() で生成する。

上記のスタートコマンドは、HTTP の POST メソッドにより送信されますが、HTTP の GET メソッドによるステータス取得用のコマンド GET LOG の処理について説明します。ユーザが GET LOG コマンドを発行すると全コンポーネントのステータスを XML メッセージにしてユーザへ送信します。

1. ログ取得コマンドに対応した関数 cb_command_log() が呼ばれる。
2. 関数 cb_command_log() から DaqOperator::command_log() が呼ばれる。
3. DaqOperator::command_log() で各コンポーネントのステータスを取得した後、Fatal エラーがなければ CreateDom::getLog() で XML のメッセージを生成する。メッセージの具体例は表 6 を参照。
4. コンポーネントのステータスに Fatal エラーがあった場合は、標準エラー出力にエラーメッセージを出力して、エラーを記述した XML のメッセージを生成する。

”Web モード”による Web ブラウザによるラン・コントロール・パネルやオンライン・ヒストグラムの実装については参考文献 [8] を参照してください。

4.7 標準入力からのコマンドによるランコントロール機能

DAQ オペレータを起動する際に”コンソールモード”または”Web モード（デフォルト）”を指定します。デフォルトは”Web モード”です。起動後にモードを変更することはできません。”コンソールモード”は、システムのデバッグやコンポーネントのテスト等の際に使用します。DAQ オペレータを起動した端末の標準入力からコマンドに対応した数字を入力してランを制御します。select() システムコールにより標準入力を監視して、（現在の実装では）2 秒間入力がなかった場合は端末の標準出力に起動したコンポーネントの名前とそのステータスを表示します。”コンソールモード”によるランコントロールの具体的な方法は、参考文献 [6] の「6 Skeleton コンポーネントによる状態遷移の確認」をご覧ください。

5 さいごに

OpenRTM-aist のバージョンアップや DAQ ミドルウェアの改良により本解説書で説明した内容が古くなる可能性があります。

参考文献

[1] 産業技術総合研究所 知能システム研究部門 OpenRTM-aist の公式 Web サイトを参照。
`http://www.openrtm.org/OpenRTM-aist/`

[2] Robotic Technology Component (RTC),Version 1.0
http://www.omg.org/spec/RTC/1.0/

[3] Y. Yasu, et al., Feasibility of data acquisition middleware based on robot technology, CHEP06, 2006.
`http://daqmw.kek.jp/docs/chep06-ID192-paper.pdf`

[4] 仲吉一男、安 芳次、千代浩司、「DAQ ミドルウェア概要」、2008 年 11 月
`http://daqmw.kek.jp/docs/daqmw-overview.pdf`

[5] 長瀬雅之、中本啓之、池添明宏、「はじめてのコンポーネント指向ロボットアプリケーション開発」、毎日コミュニケーションズ、2008 年、ISBN978-4-8399-2900-8.

[6] 千代浩司、「DAQ-Middleware 1.1.0 開発マニュアル」、2011 年 6 月
`http://daqmw.kek.jp/docs/DAQ-Middleware-1.1.0-DevManual.pdf`

[7] 安 芳次、「Condition データベースの開発マニュアル」、2009 年 7 月
`http://daqmw.kek.jp/docs/ConditionDevManual.pdf`

[8] 安 芳次、「WEB を用いた DAQ ミドルウェア GUI 開発マニュアル」、2009 年 7 月
`http://daqmw.kek.jp/docs/WebDAQGUI.pdf`

A XML/HTTP プロトコル

DAQ ミドルウェアと外部システムとの通信に使用する XML/HTTP を表 4、表 5、表 6 に示します。

コマンド	要求/応答	メソッド	URI	HTTP ボディ
CONFIGURE	要求	POST	http://xxx/daq/operatorPanel/daq.py/Params	cmd="<?xml version="1.0" encoding="UTF-8" ?> <request> <params>config.xml</params> </request>"
	応答			<?xml version="1.0" encoding="UTF-8" ?> <response> <methodName>Params</methodName> <returnValue> <result> <status>OK</status> <code>0</code> <className/> <name/> <methodName/> <messageEng/> <messageJpn/> <result> </returnValue> </returnValue> </response>
UNCONFIGURE	要求	POST	http://xxx/daq/operatorPanel/daq.py/ResetPar	
	応答			<?xml version="1.0" encoding="UTF-8" ?> <response> <methodName>ResetParams</methodName> <returnValue> <result> <status>OK</status> <code>0</code> <className/> <name/> <methodName/> <messageEng/> <messageJpn/> <result> </returnValue> </response>

表 4: 外部システムとの通信に使用する XML/HTTP(1)

コマンド	要求/応答	メソッド	URI	HTTP ボディ
START	要求	POST	http://xxx/daq/operatorPanel/daq.py/Begin	cmd="<?xml version="1.0" encoding="UTF-8" ?> <request> <runNo>1</runNo> </request>"
	応答			<?xml version="1.0" encoding="UTF-8" ?> <response> <methodName>Begin</methodName> <returnValue> <result> <status>OK</status> <code>0</code> <className/> <name/> <methodName/> <messageEng/> <messageJpn/> <result> </returnValue> </response>
STOP	要求	POST	http://xxx/daq/operatorPanel/daq.py/End	
	応答			<?xml version="1.0" encoding="UTF-8" ?> <response> <methodName>End</methodName> <returnValue> <result> <status>OK</status> <code>0</code> <className/> <name/> <methodName/> <messageEng/> <messageJpn/> <result> </returnValue> </response>

表 5: 外部システムとの通信に使用する XML/HTTP(2)

コマンド	要求/応答	メソッド	URI	HTTP ボディ
PAUSE	要求	POST	http://xxx/daq/operatorPanel/daq.py/Pause	
	応答			<?xml version="1.0" encoding="UTF-8" ?> <response> <methodName>Pause</methodName> <returnValue> <result> <status>OK</status> <code>0</code> <className/> <name/> <methodName/> <messageEng/> <messageJpn/> <result> </returnValue> </response>
RESUME	要求	POST	http://xxx/daq/operatorPanel/daq.py/Restart	
	応答			<?xml version="1.0" encoding="UTF-8" ?> <response> <methodName>Restart</methodName> <returnValue> <result> <status>OK</status> <code>0</code> <className/> <name/> <methodName/> <messageEng/> <messageJpn/> <result> </returnValue> </response>
GET LOG	要求	GET	http://xxx/daq/operatorPanel/daq.py/Log	
	応答			<?xml version="1.0" encoding="UTF-8" ?> <response> <methodName>Restart</methodName> <returnValue> <result> <status>OK</status> <code>0</code> <className/> <name/> <methodName/> <messageEng/> <messageJpn/> <result> <logs> <log> <compName>READER</compName> <state>RUNNING</state> <eventNum>100</eventNum> <compStatus>WORKING</compStatus> </log> <log> <compName>MONITOR</compName> <state>RUNNING</state> <eventNum>100</eventNum> <compStatus>WORKING</compStatus> </log> </logs> </returnValue> </response>

表 6: 外部システムとの通信に使用する XML/HTTP(3)

B DAQ コンポーネントの実装に使用する関数一覧

名前

init_command_port - コマンドポート初期化

書式

int init_command_port()

説明

コマンドポートの初期化を行う。p.11 3.2.1

返り値

常に 0 を返す

エラー

名前

init_state_table - 状態遷移テーブルの初期化

書式

void init_state_table()

説明

コンポーネントの状態遷移テーブルの初期設定を行う。11 3.2.1

返り値

なし

エラー

名前

set_comp_name - コンポーネント名設定

書式

int set_comp_name(char* name)

説明

コンポーネント名を設定する。11 3.2.1

返り値

常に 0 を返す

エラー

名前

inc_sequence_num - シーケンス番号をインクリメントする

書式

```
int inc_sequence_num()
```

説明

コンポーネントで期待されている処理が成功したら、シーケンシャル番号を 1 つ増加させる。例えば、データ読み出しコンポーネントが検出器からのデータを読み出し、後段のコンポーネントへ送信が完了した場合等。p.11 3.2.2

返り値

常に 0 を返す

エラー

名前

reset_sequence_num - シーケンス番号をリセットする

書式

```
int reset_sequence_num()
```

説明

シーケンス番号を 0 にする。p.11 3.2.2

返り値

常に 0 を返す

エラー

名前

get_sequence_num - シーケンス番号の取得

書式

```
unsigned long long get_sequence_num()
```

説明

現在のシーケンス番号を取得する。p.11 3.2.2

返り値

現在のシーケンス番号

エラー

名前

inc_total_data_size - 取得データサイズをインクリメントする

書式

```
int inc_total_data_size(unsigned int byteSize)
```

説明

これまで取得したデータサイズに指定したサイズを加える。p.12 3.2.3

返り値

常に 0 を返す

エラー

名前

reset_total_data_size - 取得データサイズをリセットする

書式

```
int reset_total_data_size()
```

説明

これまで取得したデータサイズを 0 にする。p.12 3.2.3

返り値

常に 0 を返す

エラー

名前

get_total_data_size - 取得データサイズの取得

書式

```
unsigned long long get_total_data_size()
```

説明

これまで取得したデータサイズを得る p.12 3.2.3

返り値

これまで取得したデータバイトサイズ

エラー

名前

set_header - ヘッダデータをセットする

書式

```
int set_header(unsigned char* header, unsigned int data_byte_size)
```

説明

後段のコンポーネントへ送信するデータヘッダの設定を行う。p.12 3.2.4

返り値

常に 0 を返す

エラー

名前

set_footer - フッタデータをセットする

書式

```
set_footer(unsigned char* footer)
```

説明

後段のコンポーネントへ送信するデータフッタの設定を行う。p.12 3.2.4

返り値

常に 0 を返す

エラー

名前

check_header - ヘッダデータのチェック

書式

```
bool check_header(unsigned char* header, unsigned int received_byte)
```

説明

ヘッダデータのマジックナンバおよびデータサイズのチェックを行う。p.12 3.2.4

返り値

ヘッダのマジックナンバが正しく、かつヘッダにあるデータサイズと取得したデータサイズがが正しい場合は true をそれ意外は false を返す

エラー

ヘッダのマジックナンバが正しくない場合は、標準エラー出力にエラー情報を出力し false を返す。ヘッダにあるデータサイズと取得したデータサイズが異なる場合は、標準エラー出力にエラー情報を出力し false を返す

名前

check_footer - フッタデータのチェック

書式

bool check_footer(unsigned char* footer)

説明

フッタデータのマジックナンバおよびシーケンス番号のチェックを行う。p.12 3.2.4

返り値

フッタのマジックナンバが正しく、かつフッタにかかれているシーケンス番号と自身のシーケンス番号が一致している場合は true をそれ以外は false を返す

エラー

フッタのシーケンス番号と自身のシーケンス番号が異なる場合は、標準エラー出力に情報を出力し false を返す

名前

check_header_footer - ヘッダ、フッタデータのチェックを行う

書式

bool check_header_footer(const RTC::TimedOctetSeq& in_data,unsigned int block_byte_size)

説明

p.12 3.2.4

返り値

ヘッダ、フッタのチェックを行い正しい場合は true を返す。それ以外は fatal_error_report() を呼んでエラーを DAQ オペレータへ報告し自身はエラー（アイドル）状態でコマンド待機を行う

エラー

ヘッダ情報が正しくない場合、致命的エラー（FatalType::HEADER_DATA_MISMATCH）となる。フッタ情報が正しくない場合、致命的エラー（FatalType::FOOTER_DATA_MISMATCH）となる

名前

get_event_size - イベントサイズの取得

書式

unsigned int get_event_size(unsigned int block_byte_size)

説明

取得したデータからイベントサイズを取得する。p.12 3.2.4

返り値

取得したデータのヘッダ、フッタ情報を除いたイベントデータのサイズ

エラー

名前

check_trans_lock - ストップコマンド発行のチェック

書式

```
bool check_trans_lock()
```

説明

ストップコマンドが発行されているかチェックを行う。p.13 3.2.5

返り値

返値が true の場合は、ストップコマンドが発行されている。false の場合は発行されていない

エラー

名前

set_trans_unlock - ストップコマンド受信

書式

```
bool set_trans_unlock()
```

説明

ストップコマンドが発行されている場合、停止を行うための処理を行った後にこの関数を呼ぶ。p.13 3.2.5

返り値

なし

エラー

名前

fatal_error_report - 致命的エラーの報告

書式

```
void fatal_error_report(FatalType::Enum type, int code = -1)
```

説明

定義済みの致命的エラーが起きた場合にこの関数を呼ぶ。p.13 3.2.6

返り値

なし。DaqCompDefinedException を throw する

エラー

名前

fatal_error_report - 致命的エラーの報告

書式

```
void fatal_error_report(FatalType::Enum type, const char* desc,int code = -1)
```

説明

ユーザ定義の致命的エラーが起きた場合にこの関数を呼ぶ p.13 3.2.6

返り値

なし。DaqCompUserException を throw する

エラー

名前

check_outPort_status - OutPort ステータスのチェック

書式

```
BufferStatus check_outPort_status(RTC::OutPort<RTC::TimedOctetSeq> & outPort)
```

説明

OutPort の転送ステータスのチェックを行う。p.13 3.2.7

返り値

転送が正常に終了した場合 BUF_SUCCESS を返す。タイムアウトの場合 BUF_TIMEOUT を返す。相手のバッファがフルの場合 BUF_NOBUF を返す。それ以外のエラーは、BUF_FATAL を返す

エラー

名前

check_inPort_status - InPort ステータスのチェック

書式

```
BufferStatus check_inPort_status(RTC::InPort<RTC::TimedOctetSeq> & inPort)
```

説明

InPort の転送ステータスのチェックを行う。p.13 3.2.7

返り値

転送が正常に終了した場合 BUF_SUCCESS を返す。タイムアウトの場合 BUF_TIMEOUT を返す。相手のバッファが空の場合 BUF_NODATA を返す。それ以外のエラーは、BUF_FATAL を返す

エラー

名前

check_dataPort_connections() - OutPort の接続のチェック

書式

bool check_dataPort_connections(RTC::OutPort<RTC::TimedOctetSeq> & myOutPort)

説明

指定した OutPort の接続のチェックを行う。p.13 3.2.7

返り値

指定した OutPort が InPort と接続状態であれば true を返す。接続されていなければ false を返す

エラー

名前

check_dataPort_connections() - InPort の接続のチェック

書式

bool check_dataPort_connections(RTC::InPort<RTC::TimedOctetSeq> & myInPort)

説明

指定した InPort の接続のチェックを行う。p.13 3.2.7

返り値

指定した InPort が OutPort と接続状態であれば true を返す。接続されていなければ false を返す

エラー

C config.xsd

コンフィグレーションファイルの XML スキーマを下記に示します。

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:annotation>
        <xsd:documentation xml:lang="en">
         DAQ-Components Configuration schema for DAQ-Middleware.
         Copyright 2008 Kazuo Nakayoshi. All rights reserved.
        </xsd:documentation>
    </xsd:annotation>

    <xsd:element name="configInfo" type="ConfigInfoType" />

    <xsd:element name="daqOperator" type="DaqOperatorType" />
    <xsd:element name="daqGroups"   type="DaqGroupsType" />
    <xsd:element name="daqGroup"    type="DaqGroupType" />

    <xsd:element name="components" type="ComponentsType" />
    <xsd:element name="component"  type="ComponentType" />

    <xsd:element name="inPorts"   type="InPortsType" />
    <xsd:element name="outPorts"  type="OutPortsType" />

    <xsd:element name="params"    type="ParamsType" />

    <xsd:element name="hostAddr"  type="xsd:string" />
    <xsd:element name="hostPort"  type="xsd:string" />
    <xsd:element name="instName"  type="xsd:string" />
    <xsd:element name="execPath"  type="xsd:string" />
    <xsd:element name="confFile"  type="xsd:string" />
    <xsd:element name="startOrd"  type="xsd:string" />
    <xsd:element name="inPort"    type="InPortType" />
    <xsd:element name="outPort"   type="xsd:string" />
    <xsd:element name="param"     type="ParamType"  />

    <xsd:complexType name="InPortType">
        <xsd:simpleContent>
        <xsd:extension base="xsd:string">
        <xsd:attribute name="from" type="xsd:string"/>
        </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:complexType name="ParamType">
        <xsd:simpleContent>
        <xsd:extension base="xsd:string">
        <xsd:attribute name="pid" type="xsd:string" use="required" />
        </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

    <xsd:complexType name="InPortsType">
        <xsd:sequence minOccurs="0" maxOccurs="unbounded">
            <xsd:element ref="inPort" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="OutPortsType">
        <xsd:sequence minOccurs="0" maxOccurs="unbounded">
            <xsd:element ref="outPort" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="ParamsType">
        <xsd:sequence minOccurs="0" maxOccurs="unbounded">
            <xsd:element ref="param" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
```

```
    <xsd:complexType name="ComponentType">
        <xsd:sequence>
        <xsd:element ref="hostAddr" />
        <xsd:element ref="hostPort" />
        <xsd:element ref="instName" />
        <xsd:element ref="execPath" />
        <xsd:element ref="confFile" />
        <xsd:element ref="startOrd" />
        <xsd:element ref="inPorts"  />
        <xsd:element ref="outPorts" />
        <xsd:element ref="params"   />
        </xsd:sequence>
        <xsd:attribute name="cid" type="xsd:string"/>
    </xsd:complexType>

    <xsd:complexType name="ComponentsType">
        <xsd:sequence minOccurs="1" maxOccurs="unbounded">
        <xsd:element ref="component" minOccurs="1" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="DaqGroupType">
        <xsd:sequence minOccurs="1" maxOccurs="unbounded">
            <xsd:element ref="components" minOccurs="1" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="gid" type="xsd:string"/>
    </xsd:complexType>

    <xsd:complexType name="DaqGroupsType">
        <xsd:sequence>
            <xsd:element ref="daqGroup" minOccurs="1" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="DaqOperatorType">
        <xsd:sequence>
            <xsd:element ref="hostAddr" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="ConfigInfoType">
        <xsd:sequence>
            <xsd:element ref="daqOperator" />
            <xsd:element ref="daqGroups" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```