DAQ-Middleware 1.0.0 技術解説書

仲吉一男

KEK 素核研

2010 年 8 月

概 要

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 1.0.0** 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。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ます。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 1.0.0** 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。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(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。

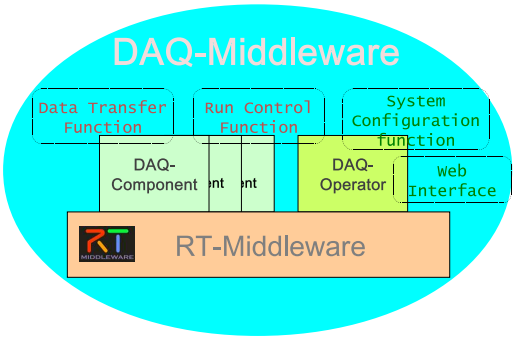

図 1: DAQ ミドルウェアと RT ミドルウェアの関係

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 1.0.0** (以降 DAQ ミドルウェア) について説明します。

図 1 に RT ミドルウェアと DAQ ミドルウェアの関係を示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。また RT コンポーネントから受け継いだ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す。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 DAQ コンポーネントへ送信します。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。

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

前述のように RT ミドルウェアにおけるソフトウェアの基本単位が 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。

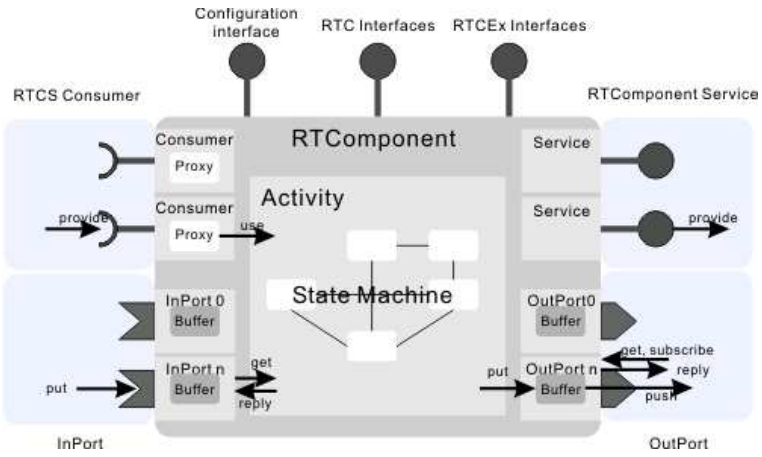

図 2: RT コンポーネントのアーキテクチャ

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、”Loaded”,”Configured”,”Running”,”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、サブコントローラを導入し、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・ファイルでは、各 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。

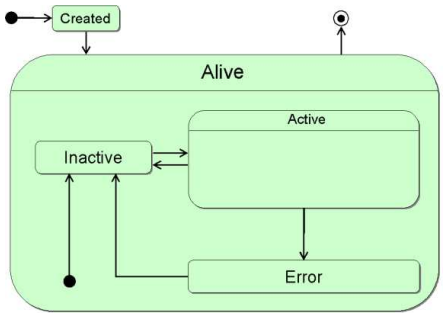

図 3: RT コンポーネントのステートチャート

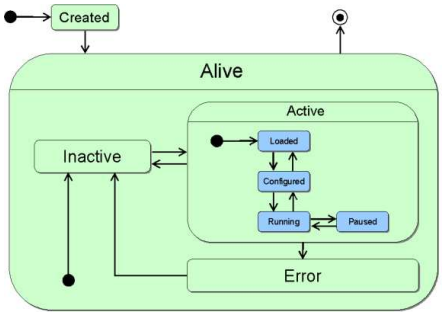

図 4: DAQ コンポーネントのステートチャート

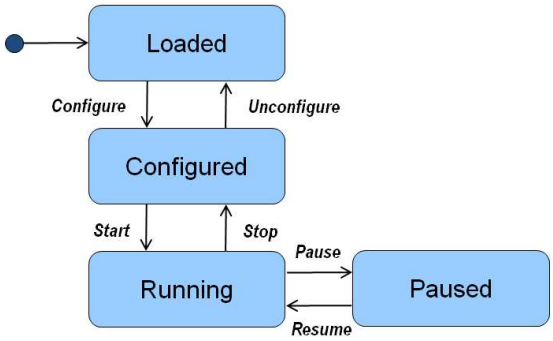

図 5: DAQ コンポーネントのコマンドと状態遷移モデル

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、同一ホスト中でもネットワーク分散環境下でも透過的にデータ転送を行うことができます。接続する際にデータポート間の通信手段を選択できます。 **DAQ-Middleware 1.0.0** では CORBA のみが選択可能です。次のリリースでは、CORBA および socket による TCP 通信の 2 つが選択可能になる予定です。socket を使ってデータ転送を行なうことで CORBA に比べてオーバヘッドが減り、データ転送性能の向上が期待できます。

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 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、それらの接続の仕方を決めることを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。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 4.1 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。システム・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「ソフトウェア・フレームワーク」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。

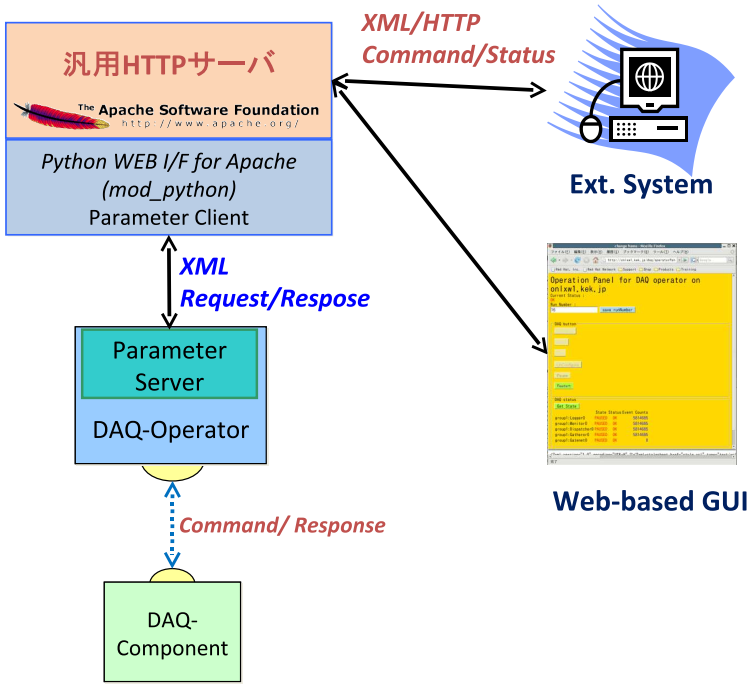

図 6: システム・インターフェイスの概念図

システム・インターフェイス機能の概念図を図 6 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。このサーバは `mod_python` により実装された”Parameter Client”からの要求を待ち受けています。`mod_python` は、Apache で Python を実行するためのモジュールです。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。

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。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 3.8 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 7 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。

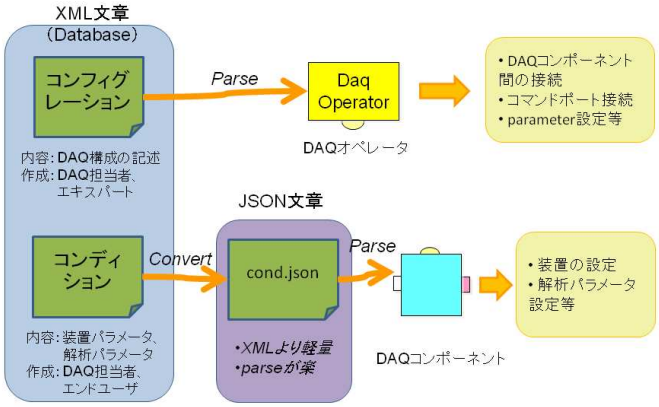

図 7: コンフィグレーション・ファイルとコンディション・ファイル

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。具体的には、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 8 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 8 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

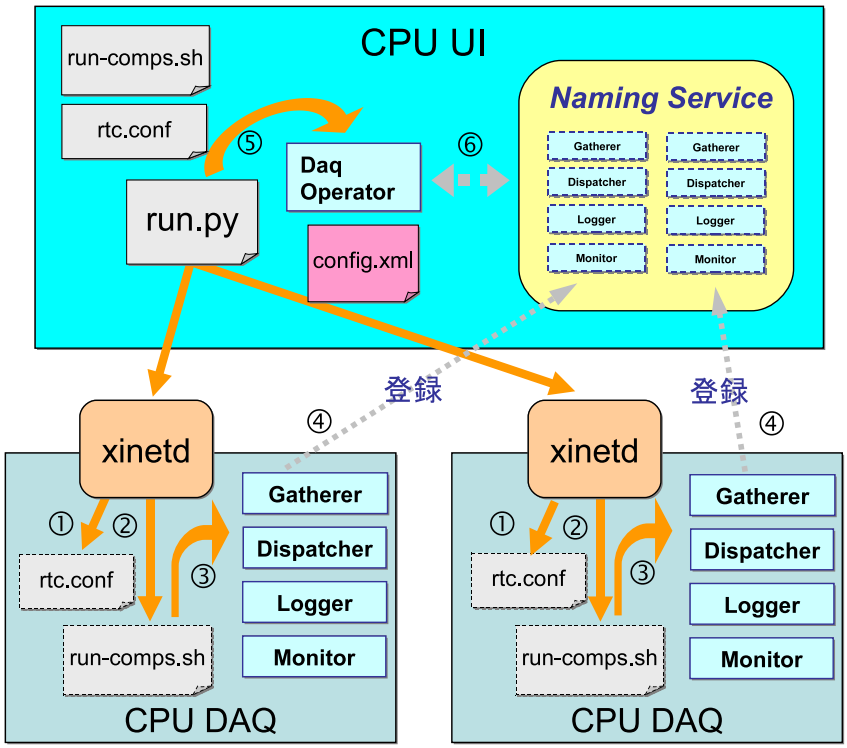

図 8: DAQ コンポーネントのリモート起動メカニズム。

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データ収集レディ状態になります。

3 DAQコンポーネントの仕様

3.1 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3.3で説明します。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9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9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。

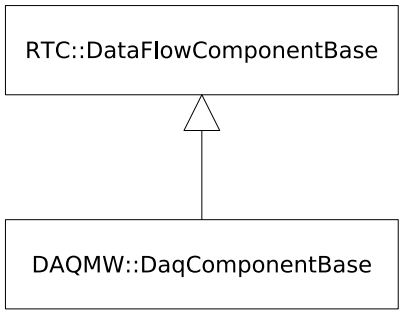

図 9: DAQコンポーネントのクラス図

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。

3.1.1 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

3.1.2 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の完了はコンポーネント開発者が知っているので、処理が完了したら `inc_sequence_num()` を使用します。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。

`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

3.1.3 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）。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

3.1.4 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, unsigned int sequence_num)`	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 値で返るので、ユーザはその値が 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ます。`check_header_footer()` では、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。

`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, unsigned loop_cnt)`	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() は与えられ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
```

3.1.5 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 3.3 で説明）がロックされます。なぜならコンポーネントが安全に 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。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
```

3.1.6 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。Fatal エラーの情報を受信したユーザ（人）または上位のフレームワークでは、ランをストップします。Fatal エラー後の処理として一般的には、ストップコマンドによりランをストップして、そのエラーの原因を取り除いて、再スタートするか全コンポーネントを終了して再起動するかのどちらかです。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 3.7 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。

3.1.7 転送ステータス取得

DAQ-Middleware 1.0.0 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}
```

DAQ-Middleware 1.0.0 では、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て、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 3.3 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;
...
```

3.2 DAQ コンポーネントのファイル構成

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 3.3 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。
- 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。
- manager->activateManager() によりコンポーネントの生成を行います。
- manager->runManager() でマネージャのメインループを実行します。 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。

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

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

    // Set module initialization proceduer
    // This procedure will be invoked in activateManager() function.
    manager->setModuleInitProc(MyModuleInit);

    // Activate manager and register to naming service
    manager->activateManager();

    // run the manager in blocking mode
    // runManager(false) is the default.
    manager->runManager();

    // If you want to run the manager in non-blocking mode, do like this
    // manager->runManager(true);

    return 0;
}
```

3.3 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 10 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。

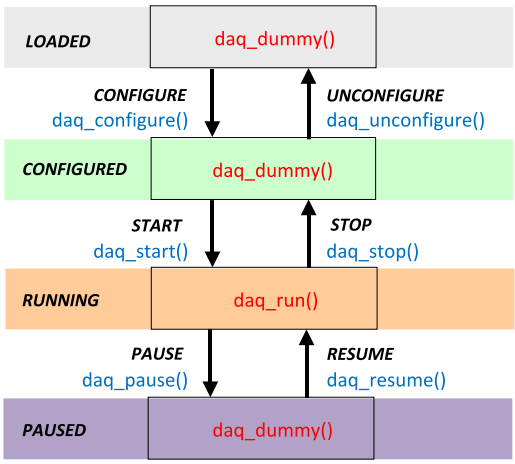

図 10: DAQ コンポーネントのステートチャート

3.3.1 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

3.4 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。

3.5 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（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;
};
```

3.6 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。

3.6.1 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()
```

3.6.2 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()
```

3.7 致命的エラー報告の送信

DAQ コンポーネントで、致命的な（Fatal）エラーが起きた場合は、fatal_error_report() という関数を呼んで 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、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。例えば、”RUNNING”状態であれば、”Stop コマンド”によりランを停止する等です。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。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。

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

3.8 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、[6] を参照してください。

4 DAQ オペレータの仕様

DAQ オペレータは、DAQ コンポーネントを制御するコンポーネントです。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）によるシステムコンフィグレーション機能
- 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
- 各 DAQ コンポーネントへのパラメータ送信機能
- 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
- 標準入力からのコマンドによるランコントロール

DAQ オペレータは、ユーザまたは上位システムからコマンドを取得して、各 DAQ コンポーネントへ通知するよう設計されています。テストまたはデバッグ用に標準入力からコマンドを取得して、同様に各コンポーネントへコマンドを通知するモードもあります。この文章ではそれぞれ「Web モード」、「コンソール・モード」と呼ぶことにします。上記のその他の機能については後述します。

4.1 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を決めるための情報

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。

4.1.1 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。

4.1.2 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。図 11 にその構造を示します。`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

4.1.3 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。この例で記述されていることを列挙します。

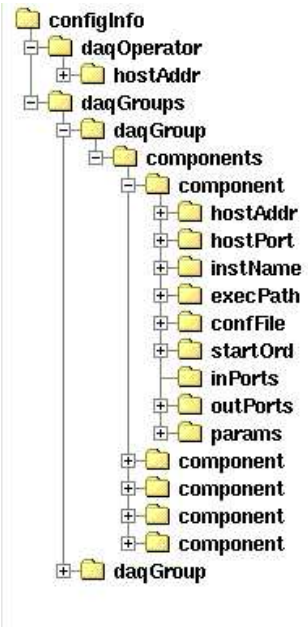

図 11: config.xml の構造

- ローカルホスト上の DAQ オペレータを使用する。
- 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。
- Reader コンポーネントは、ローカルホスト上のものを使用する。
- 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。
- Monitor コンポーネントは、ローカルホスト上のものを使用する。
- 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。
- `reader_out` と `monitor_in` は接続される。
- Reader コンポーネントの実行形式がある場所は、`/home/daq/MyDaq/Reader/ReaderComp` である。
- Monitor コンポーネントの実行形式がある場所は、`/home/daq/MyDaq/Monitor/MonitorComp` である。
- Reader コンポーネントは、`srcAddr`, `127.0.0.1`, `srcPort`, `2222` という 2 組のパラメータを持っている。
- Monitor コンポーネントは、`update_rate`, `100` という 1 組のパラメータを持っている。
- コンポーネントの起動順序は、Monitor, Reader で停止順序はその逆である。

```
<?xml version="1.0"?>
<configInfo>
    <daqOperator>
        <hostAddr>127.0.0.1</hostAddr>
    </daqOperator>
    <daqGroups>
        <daqGroup gid="group0">
            <components>
                <component cid="Reader0">
                    <hostAddr>127.0.0.1</hostAddr>
                    <hostPort>50000</hostPort>
                    <instName>Reader0.rtc</instName>
                    <execPath>/home/daq/MyDaq/Reader/ReaderComp</execPath>
                    <confFile>/tmp/daqmw/rtc.conf</confFile>
                    <startOrd>2</startOrd>
                    <inPorts>
                    </inPorts>
                    <outPorts>
                        <outPort>reader_out</outPort>
            </outPorts>
```

```
                    <params>
                        <param pid="srcAddr">127.0.0.1</param>
                       <param pid="srcPort">2222</param>
                    </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>
```

4.2 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

4.3 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 3 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。

5 さいごに

OpenRTM-aist のバージョンアップや DAQ ミドルウェアの実装の改良により、本解説書で説明した内容が古くなる可能性があります。また、現状においても説明が十分ではない箇所もありますので、今後もこの技術解説書の内容を順次更新する予定です。コメント、ご要望がありましたらお知らせください。

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] 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 http://greentea.kek.jp/daqm/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プロトコルを表4と表5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/run.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>
START	要求	POST	http://xxx/daq/operatorPanel/run.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/run.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>
UNCONFIGURE	要求	POST	http://xxx/daq/operatorPanel/run.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
PAUSE	要求	POST	http://xxx/daq/operatorPanel/run.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/run.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/run.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 5: 外部システムとの通信に使用する XML/HTTP(2)

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

p.11 3.1.1

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

p.11 3.1.1

名前

set_comp_name(char* name) - コンポーネント名設定

書式

```
int set_comp_name(char* name)
```

説明

p.11 3.1.1

名前

inc_sequence_num()

書式

```
int inc_sequence_num()
```

説明

p.12 3.1.2

名前

reset_sequence_num()

書式

```
int reset_sequence_num()
```

説明

p.12 3.1.2

名前

get_sequence_num()

書式

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

説明

p.12 3.1.2

名前

inc_total_data_size

書式

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

説明

p.12 3.1.3

名前

reset_total_data_size

書式

```
int reset_total_data_size()
```

説明

p.12 3.1.3

名前

get_total_data_size

書式

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

説明

p.12 3.1.3

名前

set_header

書式

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

説明

p.12 3.1.4

名前

set_footer

書式

```
set_footer(unsigned char* footer, unsigned int sequence_num)
```

説明

p.12 3.1.4

名前

check_header

書式

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

説明

p.12 3.1.4

名前

check_footer

書式

```
bool check_footer(unsigned char* footer, unsigned int loop_cnt)
```

説明

p.12 3.1.4

名前

check_header_footer

書式

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

説明

p.12 3.1.4

名前

get_event_size

書式

```
unsigned int get_event_size(unsigned int block_byte_size)
```

説明

p.12 3.1.4

名前

check_trans_lock

書式

```
bool check_trans_lock()
```

説明

p.13 3.1.5

名前

set_trans_unlock

書式

```
bool set_trans_unlock()
```

説明

p.13 3.1.5

名前

fatal_error_report

書式

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

説明

p.13 3.1.6

名前

fatal_error_report

書式

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

説明

p.13 3.1.6

名前

check_outPort_status

書式

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

説明

p.13 3.1.7

名前

check_inPort_status

書式

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

説明

p.13 3.1.7

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>
```