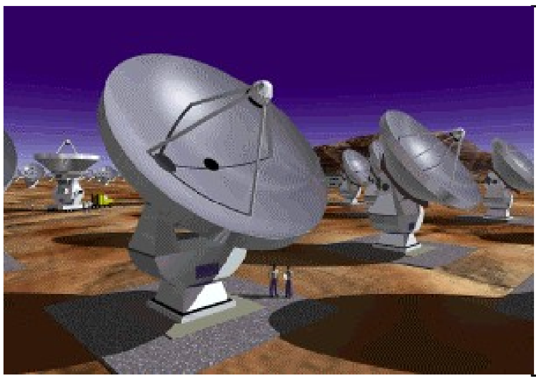

	Atacama Large Millimeter Array	ALMA-NNNNN Revision: 1.1 2016-04-21 *User Manual* Bogdan Jeram

Bulk Data NT

User Manual and How-to Manual

Bogdan Jeram
ESO

Owner	Bogdan Jeram (bjeram@eso.org)
Keywords:	
Approved by:	Date: Signature:

Change Record

REVISION	DATE	AUTHOR	SECTION/PAGE AFFECTED	REMARKS
1.0	24-07-2013	B. Jeram	All	Created.

Table of Contents

1 Overview

The ACS bulk data is a software subsystem – ACS package which allows to transfer bigger amount of data from many senders to one receiver, from one sender to many receivers, and if needed also from many senders to many receivers.

First version of the bulk data was based on the TAO Audio/Video Streaming Service, which, in turn, implements the OMG CORBA Audio/Video Streaming Service specifications. Later the bulk data was re-implemented with DDS (Data Distribution Service) technology in particular with RTI implementation of OMG DDS specification. We refer to this this version of the bulk data as to bulk data new technology, or short bulk data NT. The bulk data was designed to keep the underlying details away from the end user, what means that for end user does not have to understand the details of DDS, or A/V streaming.

At the moment the bulk data NT is implemented just in C++ programming language. However, if needed, an implementation in other programming languages like Java can be added.

Although the BDNT is based on ALMA requirements, it was designed and implemented in a way that is generic, and can be used also for other cases.

This document describes the bulk data NT and shows how to use it, configuring …

2 Design and Implementation

2.1 Basic concepts

The entity that sends data out we refer to as a **sender**, and there is a contra part entity called **receiver**, which, as name suggests, receives data. The data are transferred from the sender(s) to the receiver(s) on so called **flow**. One or more flows is/are organized inside a concept calling a **stream**. There is a sender stream, and flow and receiver stream, and flow. We refer to each stream and flow with a name. A specific data transfer path is defined with the combination of stream and flow name.

Besides bulk data defines a high level protocol for transmitting the data:

First a parameter has to be sent – a sequence of octets of arbitrary data that should be interpreted by the application that uses bulk data.
After, the data in form of one or more octet sequences can be sent.
At the end it is necessary to inform receiver(s) that no more data will be sent.
In case of bulk data NT that is based on DDS the data are transmitted in chunks – frames of a size of 64000bytes.

The implementation of Bulk Data NT comes in two flavors:

- C++ API
- ACS component (IDL) which internal uses C++ API

2.2 C++ API

For each concept described in 2.1 there exists a corresponding C++ class which is going to be described in this section.

`bulkDataNTReceiverStream` and `bulkDataNTSenderStream` are classes that represent sender or receiver stream, and they both inherits from common base class `BulkDataNTStream`. If we want to create a stream we have to insatiate an object of stream class.

Similar we have for receiver and sender flows corresponding classes: `bulkDataNTReceiverFlow` and `bulkDataNTSenderFlow.`

2.2.1 Sender stream – class `BulkDataNTSenderStream`

The sender stream class represents sender stream.

Sender stream constructor. As first parameter it takes the (sender) stream name. Optional it is possible to give configuration (`SenderStreamConfiguration`) as second parameter; otherwise a default sender stream configuration is used:

```
BulkDataNTSenderStream(const char *name, const
                       SenderStreamConfiguration &)
```

A method to create a new sender flow (`BulkDataNTSenderFlow`) on the sender stream. As first parameter it takes (receiver) flow name. Optional is possible to provide also sender flow configuration (`SenderFlowConfiguration`), a sender flow status callback (BulkDataNTSenderFlowStatusCallback), and a flag if the callback should be deleted with the flow.

```
BulkDataNTSenderFlow* createFlow(const char *flowName, const
                      SenderFlowConfiguration &cfg,
                      BulkDataNTSenderFlowStatusCallback *cb,
                      bool releaseCB)
```

An already created sender flow can be retrieved from the sender stream using flow name with the following method:

```
BulkDataNTSenderFlow *  getFlow(const char *flowName)
```

2.2.2 Sender flow - class `BulkDataNTSenderFlow`

A sender flow is represented with class `BulkDataNTSenderFlow.` A new sender flow object can be created by invoking `createFlow` method on a sender stream object (`BulkDataNTSenderStream`). The class provides three methods to send data according to the bulk data protocol:

At the beginning of the transmition we send to the receiver(s) so called "parameter" by invoking method:

```
void  startSend(const unsigned char *param, size_t len)
```

In case of a problem (timeout, or wrong order) an exception of type:

`StartSendErrorExImpl` is thrown, which contains an error trace with the details (timeout, wrong order,..) of the error.

Data we sent to the receivers(s) by calling method:
`void  `**`sendData`**`(const unsigned char *buffer, size_t len)`
one or several times. In case of a problem (timeout, wrong order – invoking `sendData` before `startSend`) an exception of type: `SendDataErrorExImpl` is thrown, which contains a detailed error trace.

We finish the transmition by invoking method:
`void  `**`stopSend`**`()`

We can get the number of so far connected receiver on the particular flow by invoking:
`unsigned int   `**`getNumberOfReceivers`**`()`

The name of the flow can be retrieved with:

`std::string    `**`getName`**`()`

The sender flow object(s) is/are destroyed when the containing sender stream object is destroyed, but it can be also destroyed explicitly by invoking C++ `delete`.

2.2.3 Sender flow status callback – class `BulkDataNTSenderFlowStatusCallback`

We can get asynchronously (via callback) informed about different status changes in the sender flow, by providing an **optional,** so called sender flow status callback object, when we create a sender flow object. In this way we can get informed about an error, and new connection/disconnection of a receiver. A user has to create a new class that derives from `BulkDataNTSenderFlowStatusCallback`) and implement one, or all of the following methods:

`virtual void `**`onError`**`(ACSErr::CompletionImpl &error)`
to get notification about an error container in an error completion.

`virtual void `**`onReceiverConnect`**`(unsigned short totalRcvs)`
to get notification about a newly connected receiver, with the total number of connected receivers.

`virtual void `**`onReceiverDisconnect`**`(unsigned short totalRcvs)`
to get notification that a receiver has disconnected, with the number of the receivers that remains connected.

The information (the name) about the stream/flow that invokes the callback can be retrieved with methods:

`const char* `**`getFlowName`**`()`

`const char* `**`getStreamName`**`()`

2.2.4 Receiver stream – class `BulkDataNTReceiverStream<>`

The receiver stream is implemented with a template class, where as the template parameter can be given a class that implements receiver callback.

Similar to sender stream we can create a receiver stream object by giving a receiver stream name, and optional receiver stream configuration (`ReceiverStreamConfiguration`):

```
BulkDataNTReceiverStream(const char *streamName,
                         const ReceiverStreamConfiguration &c)
```

There is also a possibility to specify the receiver name, so that we can distinguish between different receivers. The receiver name can be given as first parameter, the rest is the same as for the other constructor:

```
BulkDataNTReceiverStream(const char *receiverName,
                         const char *streamName,
                         const ReceiverStreamConfiguration &c)
```

The same way as for sender case we can create a new receiver flow by invoking `createFlow` method which takes as the first parameter receiver flow name. In addition can be optionally passed: a receiver flow configuration (`ReceiverFlowConfiguration`), a receiver callback, and a flag if the receiver callback has to be deleted together with the flow.

```
BulkDataNTReceiverFlow * createFlow(const char *flowName, const
                                    ReceiverFlowConfiguration
                                    &cfg, BulkDataNTCallback
                                    *cb, bool releaseCB)
```

If the (receiver) callback object is given it is used for the callbacks when data arrive, if the callback is not specified, or it is 0, an object from the template parameter is created. In this way is given a flexibility to create a callback object before with arbitrary constructor.

An already created receiver flow can be retrieved from the receiver stream using flow name with the following method:

```
BulkDataNTReceiverFlow *   getFlow(const char *flowName)
```

2.2.5 Receiver flow – class `BulkDataNTReceiverFlow`

A receiver flow is represented with class `BulkDataNTReceiverFlow.` A new receiver flow object can be created by invoking `createFlow` method on a receiver stream object (`BulkDataNTReceiverStream`). At the construction time it is possible to provide a receiver callback object, otherwise the receiver stream class creates one using class given as template parameter. The callback object can be retrieved at any time by invoking:

```
BulkDataNTCallback *    getCallbackObject()
```

or template version:

```
template<class T> T *   getCallback()
```

2.2.6 Receiver callback – class `BulkDataNTCallback`

Class BulkDataNTCallback, more precise, its implementation provides the way that the data are received, and the notification of different receiver's event like errors. For the data transmitting the three methods that reflects the bulk data protocol, have to be implemented.

```
virtual int cbStart(unsigned char *userParam_p, unsigned int
size=0
```
which is called when the parameter data arrives to the sender. This method corresponds to sender flow method `startSend`.

When the data arrives to the receiver side (to the receiver flow) the method:
```
virtual int cbReceive(unsigned char *frame_p, unsigned int
size)=0
```
is invoked. The method corresponds to the `sendData` method of the sender flow. For larger data (bigger than 64k) this method is invoked several times.

At the end of transmission it is called:
```
virtual int    cbStop ()
```

In addition the receiver callback mechanism gives to the user a possibility to get notified about possible errors:
```
virtual void   onError(ACSErr::CompletionImpl &error)
```
The method is called when an error happens in the flow's callback (`cbStart/cbReceive/cbStop`). The error is contained in the error completion.

A separate callback method that is invoked in case of the data is lost:
```
virtual void   onDataLost(unsigned long frameCount, unsigned
long totalFrames, ACSErr::CompletionImpl &error)
```
which gives to the user information about at which frame the data lost occurred(`frameCount`), and what is the total number of the occurrence(`totalFrames`).

Notification about a new sender connection, or disconnection is done by calling:
```
virtual void   onSenderConnect()
virtual void   onSenderDisconnect()
```

In the implementation of the receiver callback class can be useful to get information about the stream/flow and receiver for what can be used methods:
```
const char *   getStreamName()
const char *   getFlowName()
const char *   getReceiverName()
```

2.2.7 Configuration classes

We could see that when we create a stream or flow we can give also a configuration. For this purpose there have been introduced 4 classes:

- `SenderStreamConfiguration`
- `SenderFlowConfiguration`
- `ReceiverStreamConfiguration`
- `ReceiverFlowConfiguration`

Those classes contain member attributes per configuration. The values can be fed in different way: grammatically with setter methods, or from CDB using configuration parse what is explained in details in section: 2.4.1.

2.3 ACS component (IDL)

2.3.1 Sender component - IDL: `BulkDataSender` and C++:`BulkDataNTSenderImpl`

`BulkDataNTSenderImpl` is an abstract class which provides the implementation of the sender component - `BulkDataSender IDL.` It implements two methods:

`void` **`connect`**`(bulkdata::BulkDataReceiver_ptr receiverObj_p)`
For backward compatible reason (with A/V version of bulk data) it takes as parameter reference to a receiver, which is ignored. The method reads streams and flows information from configuration data base, and creates corresponding sender streams' and sender flows' objects.
`void` **`disconnect`**`()`
It deletes all sender streams' and flows' objects created in `connect` method.

In order to create a new sender component, the user can either create a new IDL interface which inherits from the interface `BulkDataSender` (using in this way the `connect` and `disconnect` method of the base class), or creating a completely new one. The implementation of the component/IDL can in this case inherit from `BulkDataSenderImpl` and provide the implementation of the three methods from IDL:

- **`startSend`** is used to start the data transfer (e.g. send parameters to the receiver(s), open files, etc.)
- **`paceData`** the user sends the bulk of the data to the receiver(s)
- **`stopSend`** ends the data trasfer (e.g. close the open files, etc.).

To actually send the data, the respective methods of the BulkDataNTSenderFlow class for each flow have to be called inside the three methods described above. In particular, the user has to call:

- `getSenderStream(“Stream1”)->getFlow(“FlowA”)->startSend(…)` in **`startSend`**
- `getSenderStream(“Stream1”)->getFlow(“FlowA”)->sendData(…)` in **`paceData`**
- `getSenderStream(“Stream1”)->getFlow(“FlowA”)->stopSend(…)` in **`stopSend`**

`BulkDataNTSenderStream *`**`getSenderStream`**`(const char *name)` returns the sender stream for the name (in our example “Stream1”) which has to be created in the `connect` method using information from CDB. If we have more streams in the sender component we have to “loop” over all of them.

2.3.2 Receiver component - IDL: `BulkDataReceiver` and C++:`BulkDataNTReceiverImpl< TCallback>`

The class `BulkDataNTReceiverImpl<TCallback>` implements the receiver component – `BulkDataReceiver` IDL. It implements following (IDL) methods:

void **openReceiver**()
The method reads streams and flows information from configuration data base, and creates **all** corresponding receiver streams', and receiver flows' objects.
If someone wants to "open" just particular stream and its flows that are defined in the CDB it can be used:
void **openReceiverStream**(const char *stream_name)

All the receiver streams and flows can be "closed" - destroyed by calling IDL method:
void **closeReceiver**()

If we want to "close" just a particular stream and its flows we can use:
void **closeReceiverStream**(const char *stream_name)

The Receiver can receive data only using a callback mechanism (see section 2.2.6). The template class `BulkDataNTReceiverImpl<TCallback>` provides the hook for the receiver callback. `TCallback` is a class which must inherit from `BulkDataCallback` and must implement the three methods `cbStart()`, `cbReceive()` and `cbStop()`. As it is described in section: 2.2.6: `cbStart()` is called automatically by the bulk data when the sender calls `startSend()`, `cbReceive()` when it calls `paceData()` or `sendData()`, and `cbStop()` when it calls `stopSend()`.

2.4 Configuration data base (CDB)

Primary the configuration in CDB for the bulk data NT was ment to be used for configuring stream/flows that receiver/sender bulk data component is going to create, but configuration can be used/read also directly from C++.

Sender and receiver components reads information about which streams and flows should be created from the configuration data base – CDB. There is also possible to give additional configuration like timeout, for a particular stream/flow. The configuration is given in XML format, where there is an XML element for each stream (**`SenderStream`** or **`ReceiverStream`**), which can contain one or more flow XML elements (**`SenderFlow`** or **`ReceiverFlow`**). Stream and flow configuration can be specified using XML attributes in either stream or flow element.

An example of a configuration for two streams: *Array1Stream* and *Array2Stream* with 2 flows each: *SpectralData, WVR* for a sender component:

```
<SenderStream Name="Array1Stream">
    <SenderFlow Name="SpectralData"/>
    <SenderFlow Name="WVR"/>
</SenderStream >
<SenderStream Name="Array2Stream">
    <SenderFlow Name="SpectralData"/>
    <SenderFlow Name="WVR"/>
</SenderStream >
```

and corresponding configuration for receiver component(s):

```
<ReceiverStream Name="Array1Stream">
    <ReceiverFlow Name="SpectralData"/>
    <ReceiverFlow Name="WVR"/>
```

```
                </ReceiverStream >
                <ReceiverStream Name="Array2Stream">
                    <ReceiverFlow Name="SpectralData"/>
                    <ReceiverFlow Name="WVR"/>
                </ReceiverStream >
```

As we can see we have to give to sender and receiver the same names for stream and flows to be able to establish the communication between them.

2.4.1 Reading stream/flow configuration using C++ API

If the CDB configuration is used in for defineing and configuring streams and flows inside a component the names of the stream and flows are equal to the corresponding names in the CDB. We can say that the stream/flow name and the configuration name are bound together.
With the new bulk data it is possible to distinguish between stream/flow name and the configuration name. For this purpose it was added functionality to easy use the CDB configuration also directly from C++ API. In such a way we can for example define a default configuration(s) for streams/flows that can be used in someone's application/component.
As described in section 2.2.7 there are defined 4 classes which instance represents configuring for a stream/flow. The configuration can be feed by invoking proper setter method(s), or it can be read from CDB.

The best is to look into an example how to retrieve a (default) configuration = how to use the C++ API:
Let's assume that we have in CDB (alma branch) under folder `DefaultCorrBDNTCfg` in file `DefaultCorrBDNTCfg.xml` following configuration:

```
...
<ReceiverStream Name="DefaultCORRStreamCfg">
        <ReceiverFlow Name="DefaultCORRSpectralDataFlowCfg"
           cbReceiveProcessTimeoutSec="0.02"
        >
        </ReceiverFlow>
        <ReceiverFlow Name="DefaultCORRXYZFlowCfg"/>
  </ReceiverStream>
..
```

and we would like to use this configuration for our stream(s) and flow(s).
First we have to obtain/read the (XML) data, in the standard way from the CDB:

```
CDB::DAL_ptr dal_p = getContainerServices()->getCDB();
ACE_CString CDBpath="alma/";
CDBpath +=   "DefaultCorrBDNTCfg"; //node name in CDB like
char *xmlNode = dal_p->get_DAO(CDBpath.c_str());
```

So now we have the configuration XML string from CDB which should be fed to the parser:

```
BulkDataConfigurationParser *parser_m =
new BulkDataConfigurationParser( "DefaultCorrBDNTCfg" );
//if you want to have more than one BulkDataConfigurationParser (=more
than one cfg node in CDB), should each have unique name / could be
component name.
parser_m->parseReceiverConfig(xmlNode);   //xmlNode can be also a
XML string
```

Now, we have to obtain proper the configuration from the parser, in this case for ReceiverStream (`ReceiverStreamConfiguration`) and ReceiverFlow (`ReceiverFlowConfiguration`), but it is equivalent for the sender side (`SenderStreamConfiguration` / `SenderFlowConfiguration`)

```
ReceiverStreamConfiguration*  rcvStrCfg =
parser_m->getReceiverStreamConfiguration("DefaultCORRStreamCfg");
// get Receiver Stream cfg
// now we create receiver stream with the configuration
BulkDataNTReceiverStream<MyCallback> *stream =
new BulkDataNTReceiverStream<MyCallback>("MyStream", *rcvStrCfg);
```

And similar we get the configuration for the ReceiverFlow:

```
ReceiverFlowConfiguration *rcvSpecDataFlowCfg =
parser_m->getReceiverFlowConfiguration("DefaultCORRStreamCfg",
"DefaultCORRSpectralDataFlowCfg");

BulkDataNTReceiverFlow * specDataFlow =
stream->createFlow("SpectralDataFlow", *rcvSpecDataFlowCfg);
```