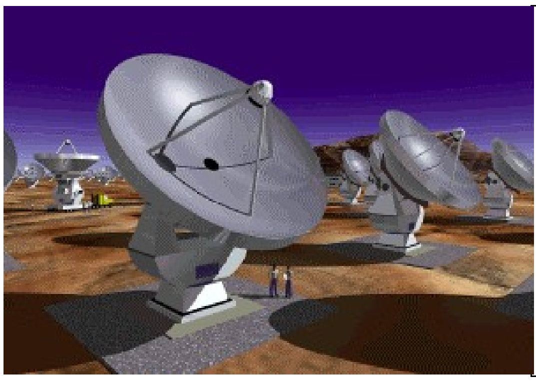


	Atacama Large Millimeter	ALMA-NNNNN Revision: 1.1 2009-04-30 *User Manual* Roberto Cirami

ACS Bulk Data Transfer

User Manual and How-to Manual

Roberto Cirami

INAF-Astronomical Observatory of Trieste

Paolo Di Marcantonio

INAF-Astronomical Observatory of Trieste

Owner	Roberto Cirami (cirami@ts.astro.it)	
Approved by:	Date:	Signature:

Change Record

REVISION	DATE	AUTHOR	SECTION/PAGE AFFECTED	REMARKS
1.0	11-11-2003	R. Cirami	All	Created.

Table of Contents

1 Overview

The ACS Bulk Data Transfer allows to transfer huge amounts of data between two ACS Components bypassing the ORB and using an out-of-bound connection for enhancing performances. In this way the achieved throughput is of the order of 800 Mbits/sec on a 1 Gbit Ethernet network (see ref. 1). The ACS Bulk Data Transfer is based on the TAO Audio/Video Streaming Service, which, in turn, implements the OMG CORBA Audio/Video Streaming Service specifications. This document describes the ACS Bulk Data Transfer and shows how to use it.

1.1 Synopsis of components in the CORBA A/V Streaming Service

The CORBA A/V Streaming Service specification defines **flows** as a continuous transfer of media between two (multimedia) devices.

Each of these flows is terminated by a **flow endpoint**.

A set of flows, such as audio flow, video flow and data flow, constitute a **stream**, which is terminated by a **stream endpoint**.

A stream endpoint can have multiple flow endpoints.

One flow endpoint acts as a source of the data and the other flow endpoint acts as a sink (see Fig. 1).

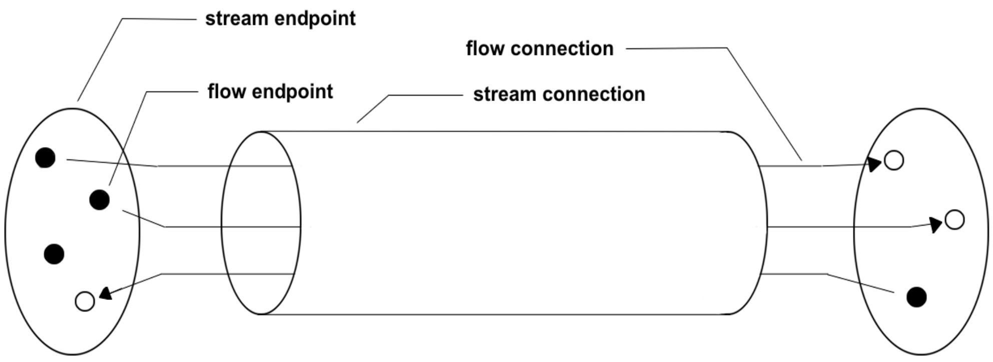


Fig. 1 Main components in the CORBA A/V Streaming Service

The data stream uses out-of-band streams, external to the GIOP/IIOP-path of the ORB, which can be implemented using communication protocols like TCP, UDP, etc., enhancing in this way the performances of the data transfer.

1.2 ACS Bulk Data Transfer features

The ACS Bulk Data Transfer module provides classes and ACS Characteristic Components which implement the features described in the above sections. It allows to create dynamically as many flows as required, specifying inside the CDB the number of flow endpoints requested (see CDB configuration section). It allows to connect a Sender component with a Receiver one, hiding all the underlying TAO A/V complexity, and provides the necessary tools to mimic a multicast behaviour (a Distributer model which connects multiple receivers to one sender).

The first sections of the document illustrate the design and the corresponding implementation; potential users who only wish to use the Bulk Data can safely skip those sections and read the user manual part of the document.

2 Design and Implementation

2.1 Requirements

The basic requirements for the ACS Bulk Data Transfer are:

- Point-to-point data transfer. Data are sent from a Sender to a Receiver using an out-of-bound high performance connection
- Multicast data transfer. A Sender sends data to a Distributer, which is responsible to deliver the data to one on more Receivers registered into it

2.2 Design

The ACS Bulk Data Transfer provides a wrapper and an adaptation of the CORBA A/V Streaming Service (in the TAO implementation) to ACS, hiding from the user most of its complexity. In order to fulfill the above requirements, C++ classes have been created, which act as wrappers around the TAO A/V Streaming Service and expose the functionalities of the TAO A/V with a simplified interface (at present only C++ implementation is provided). At a higher level, ACS Characteristic Component objects are provided, which contain and use these C++ classes, providing the developer with user-friendly programming interfaces.

These interfaces allow the user to create a stream and add to it as many flows as needed, establishing a connection between the Sender and the Receiver objects. After that the Sender can start sending data to the Receiver and then, when all the data are sent, close the connection.

2.2.1 Point-to-point

Following the CORBA A/V Streaming Service specifications, the Sender can send data in two possible ways, either asynchronously or synchronously. In the asynchronous case the data are sent inside a callback called by the TAO A/V with a timing defined by the user. The synchronous case is blocking; the data are sent directly inside the sender implementation code, without the use of a callback mechanism.

The Receiver can receive data only in an asynchronous way, by means of a callback called automatically by the TAO A/V. To reflect this behavior inside the ACS Bulk Data Transfer, the ACS Characteristic Component objects relative to the Sender and the Receiver are implemented as template classes, allowing in this way to specify the appropriate callbacks as template parameters.

2.2.2 Distributer

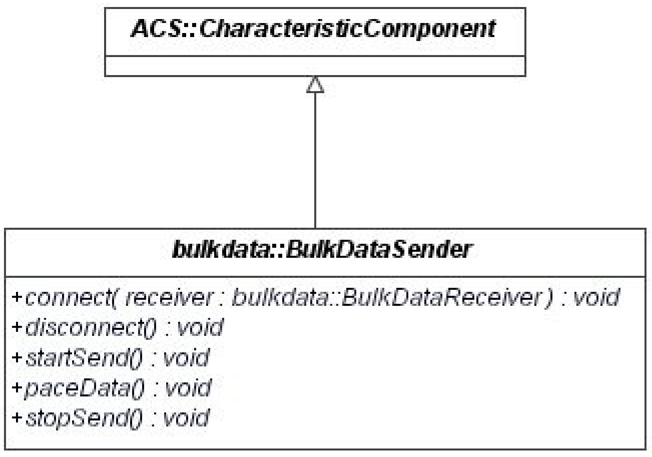
Besides a point-to-point communication model, the ACS Bulk Data Transfer provides a multicast model. Since the TCP protocol does not support multicasting a distributer mechanism has been developed. The ACS Bulk Data Distributer mimics a multicast behaviour, receiving data from one Sender and dispatching them to one or more registered Receivers.

The next subsections describe the architecture and the classes of the ACS Bulk Data Transfer. We describe in detail only the usage of the ACS Characteristic Component objects. For a complete description of the C++ classes see the online documentation.

2.3 Class description

2.3.1 Sender – class BulkDataSenderImpl<TSenderCallback>

Fig. 2 and Fig. 3 show, respectively, the relationship between the idl interfaces and the classes which form the Sender.



F

Fig. 2: Class diagram for Sender idl interfaces

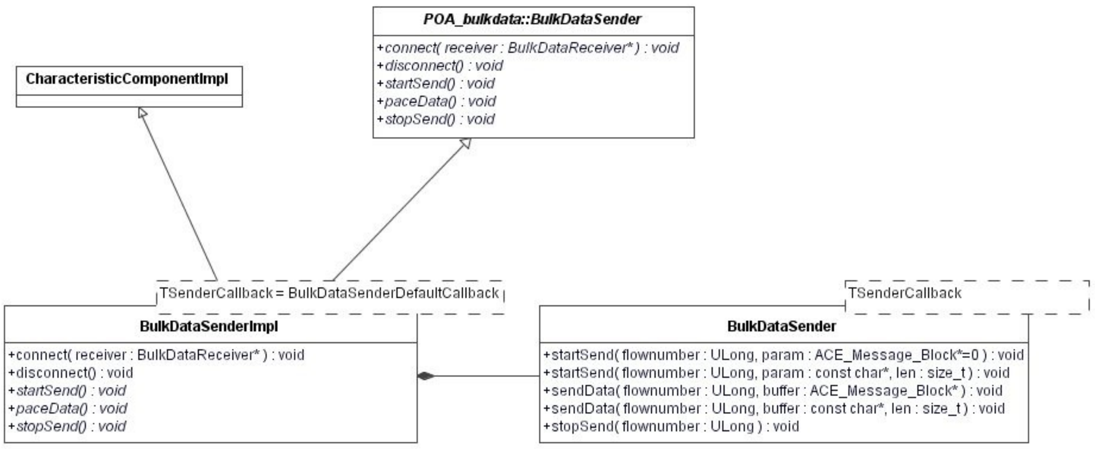

POA_bulkdata::BulkDataSender
+connect(receiver : BulkDataReceiver*) : void
+disconnect() : void
+startSend() : void
+paceData() : void
+stopSend() : void
CharacteristicComponentImpl
TSenderCallback = BulkDataSenderDefaultCallback
BulkDataSenderImpl
+connect(receiver : BulkDataReceiver*) : void
+disconnect() : void
+startSend() : void
+paceData() : void
+stopSend() : void
TSenderCallback
BulkDataSender
+startSend(flownumber : ULong, param : ACE_Message_Block*=0) : void
+startSend(flownumber : ULong, param : const char*, len : size_t) : void
+sendData(flownumber : ULong, buffer : ACE_Message_Block*) : void
+sendData(flownumber : ULong, buffer : const char*, len : size_t) : void
+stopSend(flownumber : ULong) : void

Fig. 3: Class diagram for the Sender implementation

`BulkDataSenderImpl<TSenderCallback>` is an abstract class which provides the implementation for the methods `connect` and `disconnect`. The `connect` method is responsible for the connection establishment with a Receiver object, specified as parameter. It initializes the TAO A/V and then reads from the CDB the connection parameters for the Sender: number of flows of the stream, protocol (currently only TCP) and host and port number for each flow (see CDB configuration section). After that it creates the stream endpoint and the flow endpoints, and then adds the flow endpoints to the stream endpoint. At the end it calls the `openReceiver` method on the Receiver (see description of the Receiver class) and creates the connection with it. The `disconnect` method simply closes the connection with the Receiver.

In order to implement his/her own Sender, the user can either create an idl interface which inherits from the interface `BulkDataSender` (using in this way the `connect` and `disconnect` method of the base class), ore creating a new one (not described). If his/her interface inherits from BulkDataSender, he/she must implement it in a new class which inherits from `BulkDataSenderImpl<TSenderCallback>` and provide the implementation of the `startSend`, `paceData` and `stopSend` methods (see User Manual section). `startSend` is used to start the data transfer (e.g. send parameters to the Receiver, open files, etc.). With `paceData` the user sends the bulk of the data to the receiver and `stopSend` ends the data trasfer (e.g. close the open files, etc.). To actually send the data, the respective methods of the BulkDataSender class must be called inside the three methods described above. In particular, he/she must call:

- `getSender()->startSend` in `startSend` (`getSender()` returns the private member `BulkDataSender` of the class `BulkDataSenderImpl`)
- `getSender()->sendData` in `paceData`
- `getSender()->stopSend` in `stopSend`

Fig. 4 shows the class diagram for the idl interfaces for different senders. For a complete example see the User Manual section, and in particular Example 1, files bulkDataSenderEx1.idl and bulkDataSenderEx1Impl.h (.cpp).

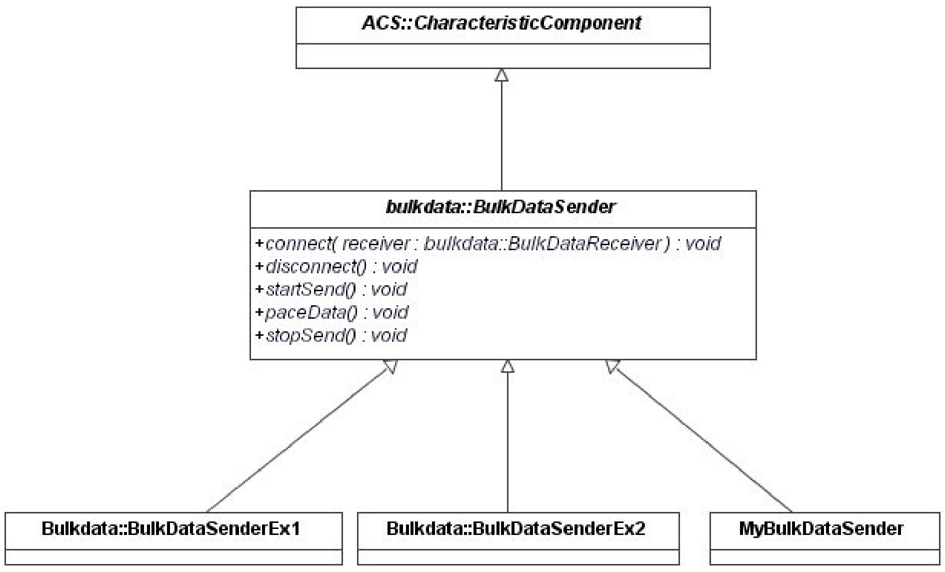

ACS::CharacteristicComponent
bulkdata::BulkDataSender
+connect(receiver : bulkdata::BulkDataReceiver) : void
+disconnect() : void
+startSend() : void
+paceData() : void
+stopSend() : void
Bulkdata::BulkDataSenderEx1
Bulkdata::BulkDataSenderEx2
MyBulkDataSender

Fig. 4: Class diagram for user Sender idl interfaces

The Sender can send data either in a synchronous or in an asynchronous way (see section 2.2). To allow the user to send asynchronous data, `BulkDataSenderImpl<TSenderCallback>` is a template class. The class which implements a Sender callback must inherit from `TAO_AV_Callback` and implement the `get_timeout(ACE_Time_Value *&timeout, void *&args)` and the `handle_timeout()` methods. The TAO A/V always calls the `get_timeout` method inside the class that implements the Sender callback. If the `timeout` parameter is zero, it means that the Sender callback is not used. If the value of `timeout` is different from zero, the TAO A/V calls the `handle_timeout` method of the Sender callback every `timeout` miscroseconds. Inside the `handle_timeout` method the user must implement the sending of the data (for example calling `getSender()->startSend`, `getSender()->sendData`, etc.).

The `BulkDataSenderImpl<TSenderCallback>` class is provided with a default template parameter which represents a Sender callback with a zero `timeout`. In this way the Sender sends its data synchronously. Via a `typedef` this “default” sender class is called `BulkDataSenderDefaultImpl`. In this way the user who wants to inherit from this class can simply do the following:

```
class UserSenderImpl : public virtual BulkDataSenderDefaultImpl
```

whereas in the first case he/she had to specify the default sender callback explicitly:

```
class UserSenderImpl : public virtual
BulkDataSenderImpl<BulkDataSenderDefaultCallback>
```

2.3.2 Receiver – class BulkDataReceiverImpl<TCallback>

Fig. 5 and Fig. 6 show, respectively, the relationship between the idl interfaces and the classes which form the Receiver.

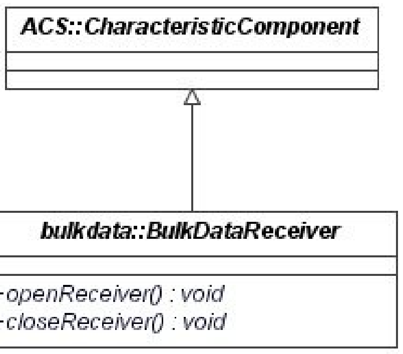


Fig. 5: Class diagram for Receiver idl interfaces

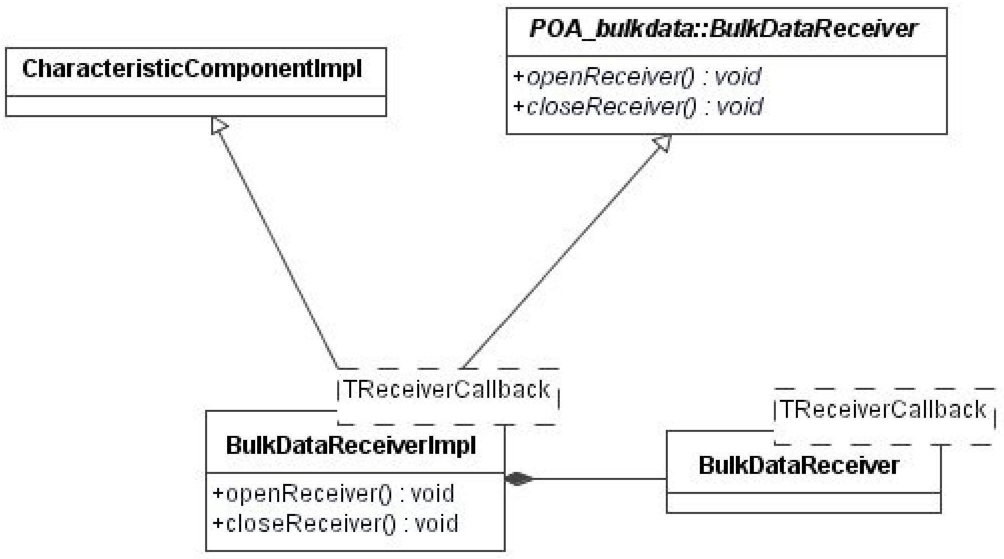


Fig. 6: Class diagram for Receiver implementation

The class `BulkDataReceiverImpl<TReceiverCallback>` provides the implementation for the methods `openReceiver` and `closeReceiver`. The `openReceiver` method, called from the Sender `connect` method, initializes the TAO A/V and then reads from the CDB the connection parameters for the Receiver: number of flows of the stream, protocol (currently only TCP) and host and port number for each flow. After that it creates the stream endpoint and the flow endpoints, and then adds the flow endpoints to the stream endpoint. The `closeReceiver` method simply closes the connection on the Receiver side.

The Receiver can receive data only using a callback mechanism (see section 2.2). The template class `BulkDataReceiverImpl<TReceiverCallback>` provides the hook for the receiver callback. `TReceiverCallback` is a class which must inherit from `BulkDataCallback` and must implement the three methods `cbStart`, `cbReceive` and `cbStop`. `cbStart` is called automatically by the TAO A/V when the Sender calls `startSend`, `cbReceive` when it calls `paceData` and `cbStop` when it calls `stopSend`.

2.3.3 Distributer – class BulkDataDistributer<TReceiverCallback, TSenderCallback>

Fig. 7 and Fig. 8 show, respectively, the relationship between the idl interfaces and the classes which form the Distributer.

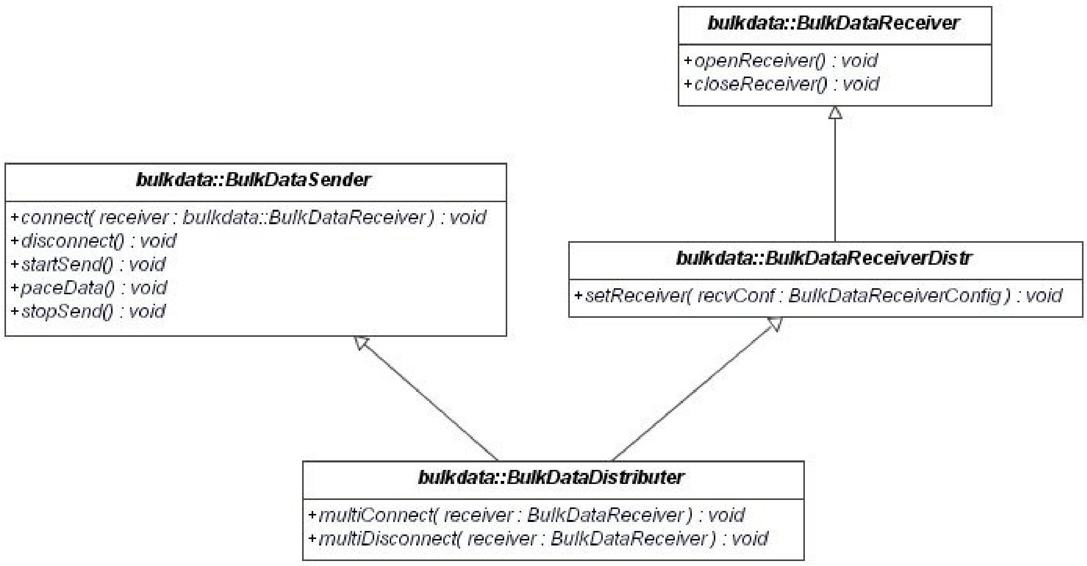

bulkdata::BulkDataReceiver
+openReceiver() : void
+closeReceiver() : void
bulkdata::BulkDataSender
+connect(receiver : bulkdata::BulkDataReceiver) : void
+disconnect() : void
+startSend() : void
+paceData() : void
+stopSend() : void
bulkdata::BulkDataReceiverDistr
+setReceiver(recvConf : BulkDataReceiverConfig) : void
bulkdata::BulkDataDistributer
+multiConnect(receiver : BulkDataReceiver) : void
+multiDisconnect(receiver : BulkDataReceiver) : void

Fig. 7: Class diagram for Distributer idl interfaces

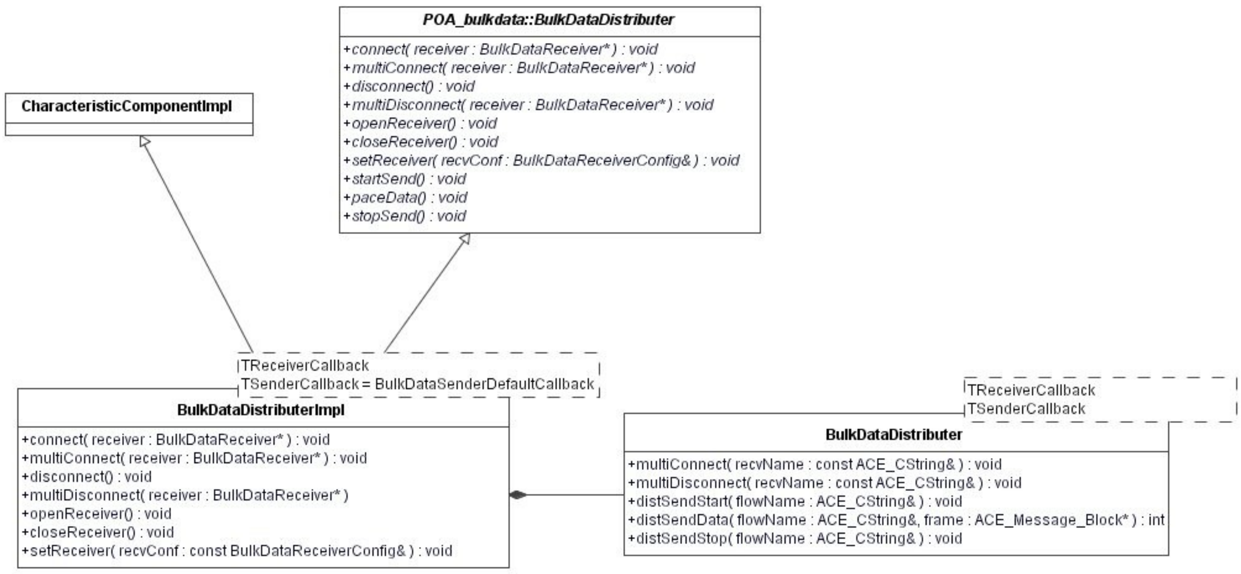

POA_bulkdata::BulkDataDistributer
+connect(receiver : BulkDataReceiver*) : void
+multiConnect(receiver : BulkDataReceiver*) : void
+disconnect() : void
+multiDisconnect(receiver : BulkDataReceiver*) : void
+openReceiver() : void
+closeReceiver() : void
+setReceiver(recvConf : BulkDataReceiverConfig&) : void
+startSend() : void
+paceData() : void
+stopSend() : void
CharacteristicComponentImpl
TReceiverCallback
TSenderCallback = BulkDataSenderDefaultCallback
BulkDataDistributerImpl
+connect(receiver : BulkDataReceiver*) : void
+multiConnect(receiver : BulkDataReceiver*) : void
+disconnect() : void
+multiDisconnect(receiver : BulkDataReceiver*)
+openReceiver() : void
+closeReceiver() : void
+setReceiver(recvConf : const BulkDataReceiverConfig&) : void
TReceiverCallback
TSenderCallback
BulkDataDistributer
+multiConnect(recvName : const ACE_CString&) : void
+multiDisconnect(recvName : const ACE_CString&) : void
+distSendStart(flowName : ACE_CString&) : void
+distSendData(flowName : ACE_CString&, frame : ACE_Message_Block*) : int
+distSendStop(flowName : ACE_CString&) : void

Fig. 8: Class diagram for the Distributer implementation

The Distributer acts as a Receiver towards the Sender and as a Sender towards the Receivers willing to get the dispatched data. So it implements both the Sender and the Receiver interfaces. The interface `BulkDataReceiverDistr` is added for internal Distributer purposes. The Sender connects to the Distributer in the usual way, with the `connect` method, passing the Distributer as parameter. The `connect` method, in turn, calls the `openReceiver` method on the Distributer (see Sender section 2.3.1).

The Distributer `multiConnect` method allows the Receivers to register themselves into the Distributer. It calls the `openReceiver` method of the Receiver passed as parameter and then calls the `multiConnect` method of the `BulkDataDistributer` object contained inside the Distributer. This method creates a new Sender internal to the Distributer and connects it to the Receiver.

When the Distributer receives parameters or data from the Sender, its receiving callback delegates to the contained `BulkDataDistributer` class the task of forwarding the incoming parameters and data to the connected Receivers via the methods `distSendStart` and `distSendData`. `distSendStop` is called whenever the Sender issues a `stopSend`.

During the disconnection phase, the `closeReceiver` method of the Distributer disconnects the Distributer itself from the Sender, whereas the `multiDisconnect` method disconnects the Distributer from the specified Receiver.

The Distributer class is a template class with two template parameters. One is the Sender callback, which takes a default value (see Sender section 2.3.1), and the other is the callback used to receive the data from the Sender and dispatch them to all registered Receivers.

2.3.4 Timeout

Inside the TAO Audio/Video Streaming Service the Sender/Receiver architecture is implemented using the ACE Reactor Pattern. Whenever the Sender sends some data, they are stored in an internal OS buffer on the receiver side. At this point the OS sends an acknowledgment to the Sender, which can send the next chunk of data. When there are data available in the receiver buffer, the ACE Reactor calls the receiver hook method which reads chunks of 8192 KB of data from the buffer. The ACE Reactor calls consecutively this hook method until there are data in the buffer. Due to this asynchronous behaviour, it could happen that the Sender receives the last acknowledgment before the data are actually processed by the user on the receiver side, causing in this way a possible data loss.

To solve this problem an ad hoc state management has been implemented inside the base receiver callback class (`BulkDataCallback`), which allows to wait automatically for the data to be processed before closing the connection (hand-shake mechanism, see Appendix A). The hand-shake mechanism sends the dimension of the transmitted data to the Receiver (in a way transparent to the user). Inside the receiver callback there is a loop and before each loop iteration the amount of the received data is compared to their expected dimension. Inside the loop there is a sleep which waits for a fixed amount of time before the next iteration. Two methods inside the `BulkDataCallback` class allow the user to set the timeout:

- `void setSleepTime(ACE_Time_Value locWaitPeriod)` sets the amount of time to wait in each loop iteration
- `void setSafeTimeout(CORBA::ULong locLoop)` sets the number of loop iterations

When the size of the received data is equal to the expected dimension, the loop ends.

3 Database configuration

It is possible to specify the number of flows inside the stream, the protocol, host and port number for the Sender and the Receiver (as well as for the Distributer) as attributes inside the CDB.

3.1.1 Sender

For the Sender, the directory …/test/CDB/alma/BulkDataSender contains the file BulkDataSender.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
   - Example Configuration Database Entry for a BulkDataSender
   -
   - Author: oat
   -
   - History:
   -   2004-11-23  oat  Created
  -->
<BulkDataSender xmlns="urn:schemas-cosylab-com:BulkDataSender:1.0"
      xmlns:baci="urn:schemas-cosylab-com:BACI:1.0"
      xmlns:cdb="urn:schemas-cosylab-com:CDB:1.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      sender_protocols="TCP=${HOST}:14000/TCP/TCP=${HOST}:14001/TCP"/>
```

The syntax of the `sender_protocols` attribute (following the A/V OMG specifications) is:

`"protocol=${HOST}:port/protocol=${HOST}:port/…/…"`

Protocol can be either TCP or UDP, and the different flows are separated by "/". In this example there are 4 flows inside the stream.

If only the protocol is specified, a default port on the localhost is assigned. Is it also possible to leave the string empty. In this case the protocol is TCP by default.

3.1.2 Receiver

The analogue attribute for the Receiver is `recv_protocols` (e. g. see …/test/CDB/alma/BulkDataReceiver/BulkDataReceiver.xml).

3.1.3 Distributer

For the Dsitributer it is necessary to specify two attributes, `recv_protocols` which specifies the number of flows towards the Sender, and `sender_protocols`, which specifies the number of flows towards the Receivers (see file …/test/CDB/alma/BulkDataDistributer.xml).

4 Error handling

The ACS Bulk Data Transfer uses local and remote exceptions to notify errors and/or abnormal program behaviour. Besides re-throwing all standard CORBA exceptions, the following ACSBulkDataError exceptions are also thrown (see file ACSBulkDataError.idl):

- ACSBulkDataError::AVFlowEndpointError
- ACSBulkDataError::AVStreamEndpointError
- ACSBulkDataError::AVStreamBindError
- ACSBulkDataError::AVSendFrameError
- ACSBulkDataError::AVCouldNotOpenFile
- ACSBulkDataError::AVObjectNotFound
- ACSBulkDataError::AVInitError
- ACSBulkDataError::AVCallbackError
- ACSBulkDataError::AVProtocolError
- ACSBulkDataError::AVInvalidFlowNumber
- ACSBulkDataError::AVConnectError
- ACSBulkDataError::AVDisconnectError
- ACSBulkDataError::AVStartSendError
- ACSBulkDataError::AVPaceDataError
- ACSBulkDataError::AVStopSendError
- ACSBulkDataError::AVOpenReceiverError
- ACSBulkDataError::AVReceiverConfigError
- ACSBulkDataError::AVCloseReceiverError

Known issue

At present it is not possible to throw exceptions from the receiver callbacks. Moreover, the receiver callbacks should not return negative values.

5 Notification mechanism

In order to inform the Sender when something happens on one or more Receivers, the Bulk Data provides a Notification mechanism. In this way a Receiver can send information about problems to the Sender or the Distributor, using a callback mechanism.

If the user wants to use the Notification mechanism, in the point-to-point model he instantiates a callback and passes its reference to the receiver subscription method. If something happens on the receiver side, the receiver callback notifies the receiver with an ACS completion containing all the necessary information. Then the receiver, automatically, calls the user callback passing back the completion.

In the distributor model, the user passes the reference of the instantiated callback to the Distributor. Then the Distributor automatically instantiates its own callback and passes its references to all the connected receivers. In case of error, the receiver callback notifies the Distributor passing a completion to it. The Distributor, in turn, calls its own callback which notifies the Sender with the completion.

What to do as a result of receiving a notification is entirely up to the Sender (stop sending the data, continue to send, etc.).

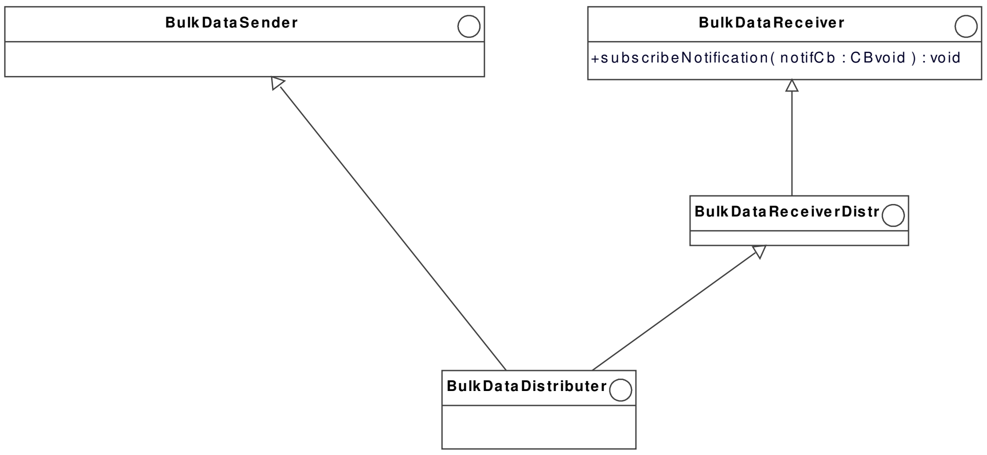


Fig. 9: Class diagram showing the relationship between the Receiver and Distributor idl interfaces.

Fig. 9 shows the interface relationship between the Receiver and Distributor. The `subscribeNotification` method of the `BulkDataReceiver` interface is used for subscribing to the Notification mechanism. This method is inherited by the `BulkDataDistributer` class.

The class `BulkDataReceiverImpl` implements the `subscribeNotification` method delegating its implementation to the `BulkDataReceiver` class (fig. 10). This class implements also the `notifySender` method necessary to notify the Sender when something happens on the receiver side.

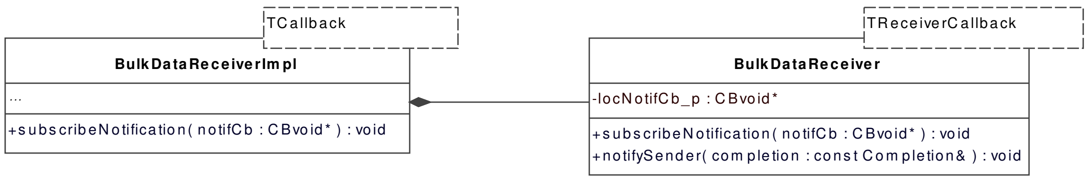

TCallback
BulkDataReceiverImpl
...
+subscribeNotification(notifCb : CBvoid*) : void
TReceiverCallback
BulkDataReceiver
-locNotifCb_p : CBvoid*
+subscribeNotification(notifCb : CBvoid*) : void
+notifySender(completion : const Completion&) : void

Fig. 10: Class diagram for the Receiver implementation.

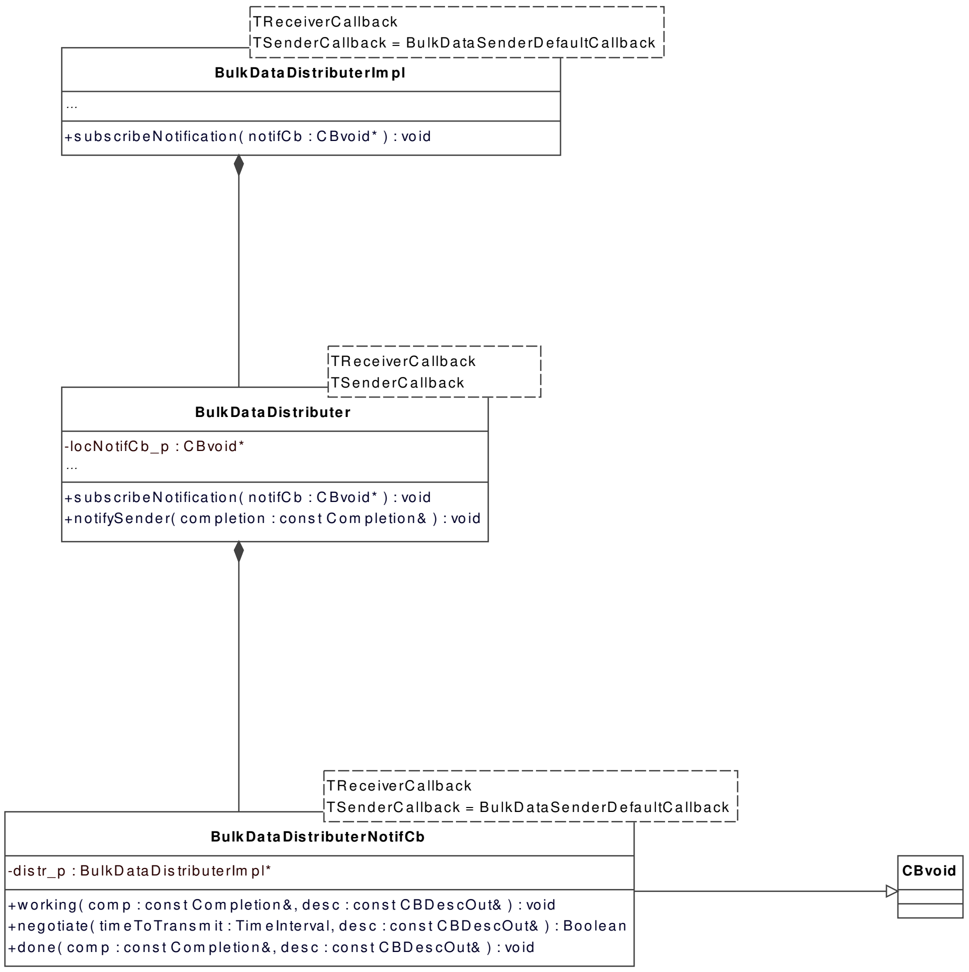

TReceiverCallback
TSenderCallback = BulkDataSenderDefaultCallback
BulkDataDistributerImpl
...
+subscribeNotification(notifCb : CBvoid*) : void
TReceiverCallback
TSenderCallback
BulkDataDistributer
-locNotifCb_p : CBvoid*
...
+subscribeNotification(notifCb : CBvoid*) : void
+notifySender(completion : const Completion&) : void
TReceiverCallback
TSenderCallback = BulkDataSenderDefaultCallback
BulkDataDistributerNotifCb
-distr_p : BulkDataDistributerImpl*
+working(comp : const Completion&, desc : const CBDescOut&) : void
+negotiate(timeToTransmit : TimeInterval, desc : const CBDescOut&) : Boolean
+done(comp : const Completion&, desc : const CBDescOut&) : void
CBvoid

Fig. 11: Class diagram for the Distributor implementation.

Also the `BulkDataDistributerImpl` class implements the `subscribeNotification` method, delegating it to the `BulkDataDistributer` class (fig. 11). This method calls automatically the `subscribeNotification` method of all the receivers connected to the Distributor, passing a reference of the Distributor Notification callback (`BulkDataDistributerNotifCb`) to them. Similarly to the receiver, also the `BulkDataDistributer` class implements the `notifySender` method.

The example 3 of section 6 shows an example of the usage of the Notification mechanism for the point-to-point and distributor models.

6 User Manual

This chapter is divided in two sections. The first one illustrates how to run the examples provided with the bulkData module. The second one shows how to customize the examples to fit the particular user needs.

6.1 ACS Bulk Data transfer examples

Inside the test directory of the bulkData module it is possible to find some examples which show the main usage of the Bulk Data Transfer.

Example 1: A synchronous Sender initializes a connection with a Receiver, establishing a stream with 4 flows. The flows perform the following operations:

Flow :1 An ACE Massage Block of 10 kB is sent from the Sender to the Receiver as parameter. Then a Message Block of 14 MB of data is transferred.

Flow :2 An array of 12000 chars is sent from the Sender to the Receiver as parameter. Then an array of 12000000 chars of data is transferred.

Flow :3 The file <…>/test/bulkDataInput.txt is read on the Sender side and then transmitted to the Receiver. The name of the output file, bulkDataOutput.txt, is sent as a parameter.

Flow :4 A Message Block of 37 MB is sent to the Receiver.

Related files for this example: bulkDataEx1.cpp, bulkDataSenderEx1Impl.h (.cpp), bulkDataReceiverCbEx1.h (.cpp).

Example 2: A Sender connects to the Distributer, sending data on two flows. The Distributer, in turn, sends data on two flows as well. Two Receivers connect to the Distributer: the first receives data on Flow1, the second on Flow2. The two Receivers are implemented in different ways, only the first one being a template class (see below). The Sender performs the following operations on the two flows:

Flow1: An ACE Massage Block of 10 kB is sent from the Sender to the Distributer as parameter. Then a Message Block of 14 MB of data is transferred.

Flow 2: The file <…>/test/bulkDataInput.txt is read on the Sender side and then transmitted to the Distributer. The name of the output file, bulkDataOutput.txt, is sent as a parameter.

Related files for this example: bulkDataDistributerEx.cpp, bulkDataSenderDistrImpl.h (.i), bulkDataReceiverDistr1Impl.h (.i), bulkDataReceiverDistr2Impl.h (.cpp), bulkDataReceiverCbDistr1.h (.cpp), bulkDataReceiverCbDistr2.h (.cpp).

Example 3: This example shows the usage of the Notification mechanism. The test is divided in two parts: in the first one a Receiver is connected to a Sender through the Distributor. The user instantiates an ACS callback and then subscribes to the Distributor passing the reference callback to it. The Sender sends some data on two flows and at the end an error is generated on flow one. Then a completion is created on the receiver side and passed back to the user callback. In the second part of the example the same notification mechanism is shown with a Receiver connected directly to the Sender (point-to-point model).

Related files for this example: bulkDataNotificationTest.cpp, bulkDataTestNotificationCb.h, bulkDataSenderNotifImpl.h (.cpp), bulkDataReceiverCbNotif.h (.cpp), bulkDataReceiver1CbNotif.h (.cpp).

Example 4: This example shows how to use different threads to send data. A Sender connects to the Distributer, sending data on four flows. The Distributer, in turn, sends the data to a Receiver. The Sender component instantiates four different threads. The same Sender object is shared between the four threads and each thread sends data on one flow, sending respectively 10, 20, 30 and 40 MB.

Related files for this example: bulkDataThreadTest.cpp, bulkDataSenderThreadImpl.h (.cpp), bulkDataReceiverCbThread.h (.cpp).

6.1.1 How to run the examples

Fist it is necessary to perform the following operations:

- export ACS_CDB=<…>/bulkData/test
- Inside the file bulkData/test/CDB/MACI/Components/Components.xml uncomment the code referring to the example you want to run and comment all the rest
- acsStart

Then the paths to follow are different for each example:

Example 1:

i. acsStartContainer –cpp ContainerEx1 in <…>/bulkData/test (the directory where the file bulkDataInput.txt resides)

ii. bulkDataEx1

Example 2:

i. acsStartContainer –cpp ContainerEx1 in <…>/bulkData/ test (the directory where the file bulkDataInput.txt resides) (Sender)

ii. acsStartContainer –cpp ContainerEx3 (Distributer)

iii. acsStartContainer –cpp ContainerEx2 (Receivers)

iv. bulkDataDistributerEx

Example 3:

i. acsStartContainer –cpp ContainerNotifTest

ii. bulkDataNotificationTest

Example 4:

iii. acsStartContainer –cpp ContainerThreadTest

iv. bulkDataThreadTest

6.1.2 How to customize the examples

This section shows in grater detail how the Bulk Data examples are built in order to allow the user to customize them to fit his needs. For brevity, of four flows only flows 1 and 3 are shown. Error handling is not included (see Error handling section).

Example 1

The test executable is bulkDataEx1.cpp. After getting the references to the Receiver and Sender, the Sender initialize the connection and then transfer parameters and data to the Receiver. At the end the connection is closed.

```
#include <maciSimpleClient.h>
#include <baci.h>
#include "ace/Get_Opt.h"
#include "orbsvcs/AV/AVStreams_i.h"
#include "bulkDataSenderC.h"
#include "bulkDataReceiverC.h"
#include "ACSBulkDataError.h"

using namespace maci;
using namespace ACSBulkDataError;

int main(int argc, char *argv[])
{
  // Creates and initializes the SimpleClient object
  SimpleClient client;
  if (client.init(argc,argv) == 0)
  {
  return -1;
  }
  else
  {
  //Must log into manager before we can really do anything
  client.login();
  }

  // Get the specific component we have requested on the command-line
  bulkdata::BulkDataReceiver_var receiver = client.get_object<bulkdata::BulkDataReceiver>("BulkDataReceiver", 0, true);

  bulkdata::BulkDataSender_var sender = client.get_object<bulkdata::BulkDataSender>("BulkDataSenderEx1", 0, true);

  sender->connect(receiver.in());

  sender->startSend();

  sender->paceData();

37:   sender->stopSend();
38:
39:   sender->disconnect();
40:
41:   receiver->closeReceiver();
42:
43:   //We release our component and logout from manager
44:   client.manager()->release_component(client.handle(),
    "BulkDataSenderEx1");
45:   ACS_SHORT_LOG((LM_INFO,"Sleeping 3 sec to allow everything to
    cleanup and stabilize"));
46:
47:   ACE_OS::sleep(3);
48:
49:   client.manager()->release_component(client.handle(),
    "BulkDataReceiver");
50:
51:   client.logout();
52:
53:   return 0;
54: }
```

5-6	idl generated files for the Sender and the Receiver
27-29	Get references to the Receiver and Sender components
33	Method used for starting the transfer and sending parameters. It calls (automatically) cbStart on the Receiver callback
35	Method used for sending the data. It calls (automatically) cbReceiver on the Receiver callback
37	Method used for ending the data transfer. It calls (automatically) cbStop on the Receiver callback
39-41	The Sender and the Receiver close the connection

Known issue: At present the disconnection order of the Sender and Receiver is important. The user **must** call `sender->disconnect()` before `receiver->closeReceiver()` in order to close the connection properly.

Sender

The first step for creating a Sender component is to create an idl interface for it which inherits from the BulkDataSender idl interface (file bulkDataSenderEx1.idl). In this new idl interface the user can add his owns methods for the Sender component.

Then it is necessary to create a class which implements this idl interface. For example 1, this class is declared inside the file bulkDataSenderEx1Impl.h:

```
1:      #include "bulkDataSenderEx1S.h"
2:      #include "bulkDataSenderImpl.h"
3:
4:      class BulkDataSenderEx1Impl : public virtual BulkDataSenderDefaultImpl,
                public virtual POA_bulkdata::BulkDataSenderEx1
5:      {
    public:

      BulkDataSenderEx1Impl(const ACE_CString& name,ContainerServices*
            containerServices);

      virtual ~BulkDataSenderEx1Impl();

      virtual void startSend()
         throw (CORBA::SystemException, AVStartSendErrorEx);

      virtual void paceData()
         throw (CORBA::SystemException, AVPaceDataErrorEx);

      virtual void stopSend()
         throw (CORBA::SystemException, AVStopSendErrorEx);
    };
```

The class BulkDataSenderEx1Impl implements a synchronous Sender, so it inherits from BulkDataSenderDefaultImpl, which provides the implementation for the `connect` and `disconnect` methods. BulkDataSenderEx1Impl must provide the implementation for the three methods `startSend`, `paceData` and `stopSend`, which are pure virtual in the base class BulkDataSenderDefaultImpl.

The implementation is provided in the file bulkDataSenderEx1Impl.cpp (error handling is not shown):

```
#include "bulkDataSenderEx1Impl.h"

BulkDataSenderEx1Impl::BulkDataSenderEx1Impl(const ACE_CString &name,
    ContainerServices* containerServices) :
    BulkDataSenderDefaultImpl(name,containerServices)
{
  ACS_TRACE("BulkDataSenderEx1Impl::BulkDataSenderEx1Impl");
}

BulkDataSenderEx1Impl::~BulkDataSenderEx1Impl()
{
  ACS_TRACE("BulkDataSenderEx1Impl::~BulkDataSenderEx1Impl");
}

void BulkDataSenderEx1Impl::startSend()
       throw (CORBA::SystemException, AVStartSendErrorEx)
{
  ACS_TRACE("BulkDataSenderEx1Impl::startSend");

  int size;

/****************************** flow 1
********************************/

  size = 10000;

  ACE_Message_Block *mb1;
  mb1 = new ACE_Message_Block(size);

  for (CORBA::Long j = 0; j < (size-1); j++)
  {
    *mb1->wr_ptr()='p';
    mb1->wr_ptr(sizeof(char));
  }

  *mb1->wr_ptr()='\0';
  mb1->wr_ptr(sizeof(char));

  CORBA::ULong flowNumber = 1;
  getSender()->startSend(flowNumber, mb1);

  ACS_SHORT_LOG ((LM_DEBUG,"flow 1 length start parameter sent = %d",
mb1->length()));

  mb1->release();

/****************************** flow 3
*******************************/

  size = 256;

  char fileName[size];

  ACE_OS::strcpy(fileName, "bulkDataOutput.txt");

  const char * ptr = fileName;

  flowNumber = 3;
  getSender()->startSend(flowNumber,ptr,size);

  ACS_SHORT_LOG ((LM_DEBUG,"flow 3 start parameter sent = %s",
fileName));

}

void BulkDataSenderEx1Impl::paceData()
     throw (CORBA::SystemException, AVPaceDataErrorEx)
{
  ACS_TRACE("BulkDataSenderImpl::paceData");

  int size;
  CORBA::ULong flowNumber;

/****************************** flow 1
*******************************/

  size = 14000000;

  ACE_Message_Block *mb1;
  mb1 = new ACE_Message_Block(size);

  for (CORBA::Long j = 0; j < (size-1); j++)
  {
   *mb1->wr_ptr()='d';
   mb1->wr_ptr(sizeof(char));
 }

 *mb1->wr_ptr()='\0';
 mb1->wr_ptr(sizeof(char));

 flowNumber = 1;
 getSender()->sendData(flowNumber, mb1);

 ACS_SHORT_LOG ((LM_DEBUG,"flow 1 length sent data = %d",
mb1>length()));

 mb1->release();

/****************************** flow 3
******************************/

 ACE_Message_Block mb(BUFSIZ);
```

```
92:
93:  FILE * fp = ACE_OS::fopen("bulkDataInput.txt","r");
94:
95: // Continue to send data till the file is read to the end.
96:  while (1)
97:  {
98:    // Read from the file into a message block.
99:    int n = ACE_OS::fread(mb.rd_ptr (),1,mb.size (),fp);
100:
101:    if (n < 0)
102:    {
103:      ACS_SHORT_LOG((LM_DEBUG," BulkDataSenderImpl<>::paceData sending
    file"));
104:      break;
105:    }
106:
107:    if (n == 0)
108:    {
109:      if (feof (fp))
110:      {
111:        // At end of file break the loop and end the client.
112:        ACS_SHORT_LOG((LM_DEBUG,"BulkDataSenderImpl<>::paceData end of
    file"));
113:        break;
114:       }
115:    }
116:
117:    mb.wr_ptr (n);
118:
119:    flowNumber = 3;
120:    getSender()->sendData(flowNumber,&mb);
121:
122:    ACS_SHORT_LOG ((LM_DEBUG,"flow 3 file bulkDatainput.txt sent"));
123:
124:    // Reset the mb.
125:    mb.reset ();
126:  } // end while
127:
128:  // Close the input file
129:  ACE_OS::fclose(fp);
130:
131: }
132:
133: void BulkDataSenderEx1Impl::stopSend()
          throw (CORBA::SystemException, AVStopSendErrorEx)
134: {
135:   ACS_TRACE("BulkDataSenderImpl::stopSend");
136:
137:   CORBA::ULong flowNumber = 1;
138:   getSender()->stopSend(flowNumber);
139:
140:   flowNumber = 3;
141:   getSender()->stopSend(flowNumber);
142: }
143:
144: /* --------------- [ MACI DLL support functions ] -----------------*/
145: #include <maciACSComponentDefines.h>
146: MACI_DLL_SUPPORT_FUNCTIONS(BulkDataSenderEx1Impl)
147: /* ----------------------------------------------------------------*/
```

13 startSend method. Used to start the transfer and to send parameters.

21-33 An ACE Message Block of 10 kB is created.

35-36	The Message Block is sent as parameter on flow 1. getSender() returns the BulkDataSender private member of the base class BulkDataSenderImpl<>.
44-50	The output filename is created.
52-53	The filename is sent as parameter on flow 3.
84	paceData method. Used to send the data.
68-83	A Message Block of 14 MB is created and sent on flow 1.
96-115	Inside the while loop the file bulkDataInput.txt is read, saved into a Message Block and then sent on flow 3.
133	stopSend method. Inside this method the transmission is stopped for each flow.

Receiver

To create the Receiver, it is necessary to provide a Receiver callback to the class BulkDataReceiverImpl<>. For example 1 the callback class is declared inside the file bulkDataReceiverCbEx1.h:

```
#include "bulkDataCallback.h"
#include "bulkDataReceiverImpl.h"

class BulkDataReceiverCbEx1 : public BulkDataCallback
{
public:
  BulkDataReceiverCbEx1();

 ~BulkDataReceiverCbEx1();

  virtual int cbStart(ACE_Message_Block * userParam_p = 0);

  virtual int cbReceive(ACE_Message_Block * frame_p);

  virtual int cbStop();

private:

  CORBA::ULong count1_m;

  CORBA::ULong count2_m;

  FILE *fp_p;
};
```

The callback class must inherit from `BulkDataCallback` and implement the three methods `cbStart`, `cbReceive` and `cbStop`. To distibguish between the different flows the internal variable flowNumber_m is used as shown in the example. The class is implemented inside the file bulkDataReceiverCbEx1.cpp:

```
#include "bulkDataReceiverCbEx1.h"

BulkDataReceiverCbEx1::BulkDataReceiverCbEx1() : count1_m(0), count2_m(0)
{
  ACS_TRACE("BulkDataReceiverCbEx1::BulkDataReceiverCbEx1");

  fp_p = 0;
}

BulkDataReceiverCbEx1::~BulkDataReceiverCbEx1()
{
  ACS_TRACE("BulkDataReceiverCbEx1::~BulkDataReceiverCbEx1");
}

int
BulkDataReceiverCbEx1::cbStart(ACE_Message_Block * userParam_p)
{
  ACS_TRACE("BulkDataReceiverCbEx1::cbStart");
  if(flowNumber_m == 1)
  {
    ACS_SHORT_LOG((LM_DEBUG,"flowname 1: %s",flowname_m.c_str()));
    ACS_SHORT_LOG((LM_DEBUG, "length param flowname 1: %d", userParam_p->length()));

    count1_m = 0;
  }
  else if(flowNumber_m == 3)
  {
    ACS_SHORT_LOG((LM_DEBUG, "flowname 3: %s", flowname_m.c_str()));
    ACS_SHORT_LOG((LM_DEBUG, "length param flowname 3: %d", userParam_p->length()));

    char filename[256];

    ACE_OS::strcpy(filename, userParam_p->rd_ptr());

    fp_p = ACE_OS::fopen (filename,"w");
  }

  return 0;
}

int
BulkDataReceiverCbEx1::cbReceive(ACE_Message_Block *
{
  ACS_TRACE("BulkDataReceiverCbEx1::cbReceive");

  if(flowNumber_m == 1)
  {
    ACS_SHORT_LOG((LM_DEBUG, "flowname 1: %s", flowname_m.c_str()));
    ACS_SHORT_LOG((LM_DEBUG, "length param flowname 1: %d", frame_p->length()));

    count1_m += frame_p->length();
  }
  else if(flowNumber_m == 3)
  {
    while (frame_p != 0)
    {
```

```
57:       int result = ACE_OS::fwrite (frame_p->rd_ptr(),
58:       frame_p->length(),1,fp_p);
59:       if (result == 0)
60:       {
61:         ACS_SHORT_LOG((LM_ERROR,"BulkDataReceiverCbEx1::cbR
   eceive failed"));
62:         return -1;
63:       }
64:       frame_p = frame_p->cont ();
65:     }
66:   }
67:
68:   return 0;
69: }
70:
71: int
72: BulkDataReceiverCbEx1::cbStop()
73: {
74:   ACS_TRACE("BulkDataReceiverCbEx1::cbStop");
75:
76:   if(flowNumber_m == 1)
77:     ACS_SHORT_LOG((LM_INFO, "flow 1 total length: %d",
   count1_m));
78:
79:   if(flowNumber_m == 3)
80:     ACE_OS::fclose(fp_p);
81:
82:   return 0;
83: }
84:
85: /* --------------- [ MACI DLL support functions ]
   ---------------*/
86: #include <maciACSComponentDefines.h>
87: MACI_DLL_SUPPORT_FUNCTIONS(BulkDataReceiverImpl<BulkDataRec
   eiverCbEx1>)
88: /*
   -------------------------------------------------------------
   -*/
```

16	The cbStart method is called (automatically) in response to the Sender startSend call. For flow 1 a counter is initialized for statistics purposes.
26-36	For flow 3, the name of the output file is extracted and the file is opened.
42	The cbReceive method is called (automatically) in response to the Sender paceData call. For flow 1 the counter variable is incremented by the length of the data received in order to calculate the total amount of data received.
53-66	For flow 3, the content of the input file is extracted and saved inside the output file.
72	The cbStop method is called (automatically) in response to the Sender stopSend call. For flow 1 the statistic is printed. For flow 3, the output file is closed.
85-88	The Receiver object is instantiated, passing as template parameter the receiver callback just implemented.

Example 2 (Distributer)

Two different ways of implementing the Receivers are used in this example. Only the first Receiver is a template.

The following code shows the relevant part of the implementation of the first receiver:

File bulkDataReceiverDistr1Impl.h:

```
template<class TCallback>
class BulkDataReceiverDistr1Impl : public virtual BulkDataReceiverImpl<TCallback>,
                          public virtual POA_bulkdatadistr::BulkDataReceiverDistr1
```

The class `BulkDataReceiverDistr1Impl` is a template class, taking the callback as template parameter. This callback is also passed as parameter to the template base class `BulkDataReceiverImpl.`

File bulkDataReceiverDistr1Impl.cpp:

```
#include <maciACSComponentDefines.h>
MACI_DLL_SUPPORT_FUNCTIONS(BulkDataReceiverDistr1Impl<BulkDataReceiverCbDistr1>)
```

The specific callback is passed when the class `BulkDataReceiverDistr1Impl` is istantiated inside the file containing the callback implementation.

The following code shows the relevant part of the implementation of the second receiver:

File bulkDataReceiverDistr1Impl.h:

```
class BulkDataReceiverDistr2Impl : public virtual
BulkDataReceiverImpl<BulkDataReceiverCbDistr2>,
                          public virtual POA_bulkdatadistr::BulkDataReceiverDistr2
```

The class BulkDataReceiverDistr2Impl is not a template, and the callback is passed directly as template parameter of the base class.

File bulkDataReceiverDistr2Impl.cpp

```
#include <maciACSComponentDefines.h>
MACI_DLL_SUPPORT_FUNCTIONS(BulkDataReceiverDistr2Impl)
```

In this case the class BulkDataReceiverDistr2Impl is not more a template and therefore is instantiated in the usual way.

Known issue: Also in this case the disconnection order is important for the correct shutdown of the connection. The two Receivers **must** be disconnected from the Distributer after the Sender:

1) `sender->disconnect()`
2) `distributer->closeReceiver()`
3) `distributer->multiDisconnect(receiver1.in())`
4) `distributer->multiDisconnect(receiver2.in())`

Example 3 (Notification mechanism)

This example shows the usage of the Notification mechanism. Only the relevant part of the code is shown in the listings below.
The test executable is bulkDataNotificationTest.cpp. The test is divided in two parts, to show the Notification mechanism usage in the distributor and point-to-point models respectively. After getting the references to the Sender, Distributor and two Receivers (not shown), the connections are established. Then the subscription to the Notification mechanism is done and the data transfer begins. At the end, all the components are released (not shown).

```
#include "bulkDataTestNotificationCb.h"

int main(int argc, char *argv[])
{
  // Here we instantiate our components

  // First part of the example (distributor model)

  sender->connect(distributer.in());
  distributer->multiConnect(receiver.in());

  BulkDataTestNotificationCb *notifCb = new
BulkDataTestNotificationCb();

  ACS::CBvoid_var cb = notifCb->_this();

  distributer->subscribeNotification(cb);

  sender->startSend();
  sender->paceData();
  sender->stopSend();

  sender->disconnect();
  distributer->closeReceiver();
  distributer->multiDisconnect(receiver.in());

  notifCb->_remove_ref();

  // Second part of the example (point-to-point model)

  sender->connect(receiver1.in());

  BulkDataTestNotificationCb *notifCb1 = new
BulkDataTestNotificationCb();

  ACS::CBvoid_var cb1 = notifCb1->_this();

  receiver1->subscribeNotification(cb1);

  sender->startSend();
39:  sender->paceData();
40:  sender->stopSend();
41:
42:  sender->disconnect();
43:  receiver1->closeReceiver();
44:
45:  notifCb1->_remove_ref();
46:
47:  //Here we release our components and logout from the
   manager
48: }
```

1	The user notification callback is included.
12-14	The user notification callback is instantiated and activated.
16	The callback reference is passed to the Distributor to subscribe to the Notification mechanism.
26	The method _remove_ref() must be called on the notification callback pointer in order to correctly decrease the reference counting and have the callback destructor called.

For the second part of the example (Distributor model) the situation is similar.

Below the code for the user notification callback is reported (bulkDataTestNotificationCb.h). The `done` method is called at the end of the notification process and the completion is logged.

```
#include <baci.h>

class BulkDataTestNotificationCb: public virtual POA_ACS::CBvoid
{
  public:

    BulkDataTestNotificationCb() {}
    ~BulkDataTestNotificationCb() {}

    void working(const Completion &comp, const ACS::CBDescOut &desc)
     throw (CORBA::SystemException)
     {}

    void done(const Completion &comp, const ACS::CBDescOut &desc)
     throw (CORBA::SystemException)
    {
      CompletionImpl complImp = comp;
      complImp.log();
    }

    CORBA::Boolean negotiate (ACS::TimeInterval timeToTransmit, const
   ACS::CBDescOut &desc)
     throw (CORBA::SystemException)
    {
      return true;
    }

};
```

In order to notify the user when something happens on the receiver side, the receiver itself must create a completion and notify the Distributor (or the Sender). This is done in the receiver callback and is shown in the listing below (file bulkDataReceiverCbNotif.cpp, only the relevant code is shown):

```
1:     #include "bulkDataReceiverCbNotif.h"
2:
3:     int BulkDataReceiverCbNotif::cbStop()
4:     {
5:       try
6:       {
7:         if(flowNumber_m == 1)
8:         {
9:            AVFlowEndpointErrorExImpl err = AVFlowEndpointErrorExImpl(__FILE__,__LINE__,"BulkDataReceiverCbNotif::cbStop");
10:           throw err;
11:        }
12:        else if(flowNumber_m == 2)
13:        {
14:           ACS_SHORT_LOG((LM_INFO, "BulkDataReceiverCbNotif - data on flow 2 total length: %d", count2_m));
15:        }
16:     }
17:     catch(ACSErr::ACSbaseExImpl &ex)
18:     {
19:        AVFlowEndpointErrorCompletion comp(ex,__FILE__, __LINE__, "BulkDataReceiverCbNotif::cbStop");
20:        comp.log();
21:        if(recv_p)
22:        {
23:          recv_p->notifySender(comp);
24:        }
25:        else
26:        {
27:          ACS_SHORT_LOG((LM_ERROR, "BulkDataReceiverCbNotif - Receiver reference NULL"));
28:        }
29:     }
30:
31:   return 0;
32:   }
33:
34:   void BulkDataReceiverCbNotif::setReceiver(AcsBulkdata::BulkDataReceiver<BulkDataReceiverCbNotif> *recv)
35:   {
36:     if (recv == NULL)
37:     {
38:        ACS_SHORT_LOG((LM_ERROR,"BulkDataReceiverCbNotif::setReceiver recv = 0"));
39:     }
40:     else
41:     {
42:        recv_p = recv;
43:     }
44:   }
```

9-10 An error is generated and thrown for test purposes inside the cbStop method.

19	In the catch block a completion is created and the error added to it.
23	The receiver is notified of the error and the completion is passed to it.
34	The method setReceiver must be implemented in order to set the receiver reference inside the receiver callback.

Example 4 (Using threads to send data)

This example shows how to use different threads to send data on different flows. Only the relevant part of the code is shown in the listings below.
The following code shows the file bulkDataSenderThreadImpl.h, where the thread class and the sender component class are declared

```
 1:     #include "bulkDataSenderDistrS.h"
 2:     #include "bulkDataSenderImpl.h"
 3:     #include "bulkDataSenderDefaultCb.h"
 4:
 5:     #include <acsThread.h>
 6:     #include <vector>
 7:
 8:     //forward declaration
 9:     class BulkDataSenderThreadImpl;
 10:
 11:  using namespace ACS;
 12:
 13:  class SenderThread : public ACS::Thread
 14:  {
 15:  public:
 16:
 17:  SenderThread(const ACE_CString& name,
BulkDataSenderThreadImpl *sender,
CORBA::ULong flowNumber,
const ACS::TimeInterval&
responseTime=ThreadBase::defaultResponseTime,
const ACS::TimeInterval&
sleepTime=ThreadBase::defaultSleepTime);
 18:  ~SenderThread();
 19:  virtual void run();

 20:  private:

 21:  BulkDataSenderThreadImpl *sender_p;
 22:  ACE_Message_Block *mb_p;
 23:  CORBA::ULong flowNumber_m;
 24:  };
 25:
 26:  class BulkDataSenderThreadImpl : public
    virtual   BulkDataSenderDefaultImpl,
                                          publi
c virtual
POA_bulkdatadistr::BulkDataSenderDistr
 27:  {
```

```
 28:
 29:  public:
 30:
 31:  BulkDataSenderThreadImpl(const
    ACE_CString& name,ContainerServices*
    containerServices);
 32:  virtual ~BulkDataSenderThreadImpl();
 33:  virtual void startSend()
      throw (CORBA::SystemException,
AVStartSendErrorEx);
 34:  virtual void paceData()
      throw (CORBA::SystemException,
AVPaceDataErrorEx);
 35:  virtual void stopSend()
      throw (CORBA::SystemException,
AVStopSendErrorEx);
 36:
 37:  private:
 38:
 39:  CORBA::ULong numberOfFlows;
 40:  vector<SenderThread*> thread_p;
 41:  };
```

13	The SenderThread class inherits from ACS::Thread
40	The sender component class BulkDataSenderThreadImpl has a vector of pointers to SenderThread as attribute.

The file bulkDataSenderThreadImpl.cpp contains the definition of the SenderThread and BulkDataSenderImpl classes. Below the relevant code related to the thread creation and the sending of data is shown.

```
1: BulkDataSenderThreadImpl::BulkDataSenderThreadImpl(const ACE_CString&
      name,ContainerServices* containerServices)
      :BulkDataSenderDefaultImpl(name,containerServices)
2: {
3: ACS_TRACE("BulkDataSenderThreadImpl::BulkDataSenderThreadImpl");
4:
5: numberOfFlows = 4;
6: CORBA::ULong flow;
7:
8: for(CORBA::ULong i = 0; i < numberOfFlows; i++)
9:   {
10:     string str = "Flow";
11:     flow = i + 1;
12:     stringstream out;
13:     out << flow;
14:     str = str + out.str();
15:     ACE_CString str1 = str.c_str();
16:     SenderThread *thread = new SenderThread(str1, this, flow,
      ThreadBase::defaultResponseTime,
      ThreadBase::defaultSleepTime*10 /*=1s*/);
17:     thread_p.push_back(thread);
18:   }
19: }
```

```
20:
21:
22: void BulkDataSenderThreadImpl::paceData()
                   throw (CORBA::SystemException, AVPaceDataErrorEx)
23: {
24: ACS_TRACE("BulkDataSenderThreadImpl::paceData");
25:
26: for(CORBA::ULong i = 0; i < numberOfFlows; i++)
27:   {
28:     thread_p[i]->resume();
29:   }
30: ACE_Time_Value tv(20);
31: BACI_CORBA::getORB()->run(tv);
32: }
33:
34:
35: void SenderThread::run()
36: {
37: try
38:   {
39:     sender_p->getSender()->sendData(flowNumber_m, mb_p);
40:     ACS_SHORT_LOG ((LM_INFO,"flow %d length sent data =
      %d",flowNumber_m,mb_p->length()));
41:   }
42: catch(...)
43:   {
44:     ACS_SHORT_LOG((LM_ERROR,"Error!"));
45:   }
46: setStopped();
47: ACS_SHORT_LOG((LM_INFO, "%s: Stopped thread", getName().c_str()));
48: }
```

1-19 In the constructor of the sender component class the four threads are instantiated.

28 In the paceData() method the threads are started.

39 The actual sending of data is implemented in the run() method of the SenderThread class. The ACE message blocks containing the data to be sent are created in the constructor of the SenderThread class (not shown).

6.1.3 Remarks on callback deletion

Instances of the callback classes are destroyed inside the *handle_destroy* method by invoking a *delete this* call (i.e. commiting a suicide). This is not completely safe and in principle should be used only if objects are allocated on the heap.

Unfortunately, we are forced to do this, because of the underlying implementation in the ACE TAO A/V.

An ACS Callback class is instantiated ***on the heap*** in the method "*get_callback*" (for details see the file *bulkData/include/bulkDataFlowConsumer.i*) This method is in turn called (automatically) inside the ACE TAO A/V (see *$ACE_ROOT/TAO/orbsvcs/orbsvcs/AV/TCP.cpp* line ~297).

The problem is that when the connection is closed, the TAO A/V (i.e. the reactor) *TAO_AV_TCP_Object* object calls automatically the method *handle_destroy* (see line 259 of *$ACE_ROOT/TAO/orbsvcs/orbsvcs/AV/TCP.cpp*) and after that, destroy itself (!!) (line 260, 261).

If the callback is not destroyed in the *handle_destroy* method, then we have a memory leak. If we try to destroy the callback later, for example in the destructor of the *bulkDataFlowConsumer* then we have a crash of the whole system, because the *TAO_AV_TCP_Object* object (which owns the callback) was already destroyed.

Shortly, we spent really lot of time to figure out all this behavior and the only solution seems to clean-up things by using the *"delete this"* idiom. Note however that the users are never supposed to instantiate callback object by themselves and therefore this should not give any major problem.

6.1.4 Remarks on the Makefile

In order to be able to compile correctly the code for the A/V the following lines must be added to the standard ACS Makefile:

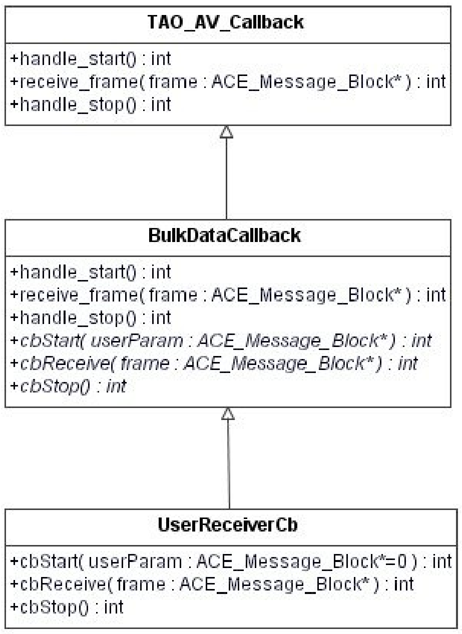
- USER_INC = -I$(TAO_ROOT)/orbsvcs/orbsvcs/AV
- USER_LIB = -lTAO_AV

It is necessary to explicitly add $(TAO_ROOT)/orbsvcs/orbsvcs/AV to the include path because some include files in the AV directory assume that -I- option is NOT used and therefore do not prepend AV/ to the file name when including it.

7 Appendix A: Hand-shake mechanism

In the TAO A/V Streaming Service, the Sender/Receiver architecture is implemented by using the ACE Reactor pattern, and uses a callback mechanism to actually manage the incoming data stream. The provided `TAO_AV_Callback` class offers three methods to fulfill this purpose: `handle_start()` and `handle_stop()`, which react when a start/stop is issued on a specific flow, and a `receive_frame(ACE_Message_block *frame)`, which is used to get the received data. This mechanism has the following limitations:

1) there is no possibility to send short parameters directly when a start is issued (for example an UID to characterize the forthcoming frame, a string containing a filename to be opened, etc.)
2) a synchronization problem occurs


TAO_AV_Callback
+handle_start() : int
+receive_frame(frame : ACE_Message_Block*) : int
+handle_stop() : int
BulkDataCallback
+handle_start() : int
+receive_frame(frame : ACE_Message_Block*) : int
+handle_stop() : int
+cbStart(userParam : ACE_Message_Block*) : int
+cbReceive(frame : ACE_Message_Block*) : int
+cbStop() : int
UserReceiverCb
+cbStart(userParam : ACE_Message_Block*=0) : int
+cbReceive(frame : ACE_Message_Block*) : int
+cbStop() : int
Fig. 12: Receiver callback class diagram

Point 2 is quite subtle. Data sent by the Sender are first received in the TCP-receive memory buffer of the involved host (whose typical default size for Linux Red Hat 9.0 is around 85 KB). Being the ACE Reactor event-driven, as soon as data are available the pre-registered callback method is called and data are consumed (the reactor concrete event handler is the `receive_frame` method, as described before). The limitation is that internally the TAO A/V reads data only in chunks of 8192 bytes. It could happen therefore that the Sender receives the acknowledgement of the last frame received even if the data are still not fully consumed on the Receiver side (they are actually stored in the host TCP receive buffer, but are not read yet). In this case a stop could be issued to early spoiling the last part of the received stream.

In order to overcome this problem, a hand-shake protocol on top of this architecture has been implemented, by inheriting from the TAO_AV_Callback (as shown in fig. 12), and adding internally a new state management (see below). Before sending the raw data, a control frame is sent and analyzed by the `BulkDataCallback` class. The control frame contains the information (an ID) on whether the forthcoming stream is a parameter or the bulk of data, and the number of expected bytes length. The ID allows to call internally the appropriate methods (`cbStart(ACE_Message_Block *param) / cbReceive(ACE_Message_Block * frame)`) to distinguish between parameters and data, whereas the bytes length information permits to manage and overcome the synchronization problem. The hand-shake mechanism checks internally if the dimension of the received data is equal to the expected data length, and waits until all data have been received (see below).

The hand-shake mechanism described above is completely hidden from the user. To receive fully synchronized parameters/data she/he must only inherit from BulkDataCallback and implement the three abstract methods (see Fig. 12), without knowing anything about what happens below.

The hand-shake mechanism introduces some performance penalties in the data throughput, bout only of the order of 0.5% (see ref. 1).

Fig. 13 shows the sequence diagram for the hand-shake mechanism when the user issues the commands `startSend()`/`paceData()`/`stopSend()`.

From the main program, the user calls `startSend()` on the Sender (point 1 in fig. 13), calling, in turn, `startSend(flwn,param)` on the BulkDataSender object (point 2), passing as parameters the flownumber and the parameter. The implementation of this last method is hidden from the user. Inside this method the hand-shake mechanism is implemented, consisted of five sequent method calls:

a) `streamctrl_p->start()`. This calls the `start()` method of the `TAO_StreamCtrl` object, which is a private member of the class `BulkDataSender`. This method calls asynchronously the `handle_start()` method of the `BulkDataCallback`, setting its internal state.

b) `dp_p->send_frame(1,dim)`. This calls the `send_frame` method of the `TAO_AV_Protocol_Object` object (private member of the class `BulkDataSender`); the first parameter, 1, indicates that the data that are going to be received are parameters, and dim is the dimension of the parameter itself. This method calls asynchronously the hook method of the `ACE_Reactor`, called when there are data in the receiving buffer. This hook method, in turn, calls the `receive_frame` method of the `BulkDataCallback`. The index 1 is passed to `receive_frame` as parameter, indicating that the data that are going to arrive are parameters.

c) `streamctrl_p->stop()`.This method calls asynchronously the `handle_stop()` method of the `BulkDataCallback`, setting its internal state.

d) `dp_p->send_frame(param)`. After the internal state of the callback has been set, the actual parameter are sent, and the `ACE_Reactor` calls `receive_frame(param)` in response to this method call. `receive_frame`, in turn, calls the `cbStart(param)` method of the user callback, passing the parameter to it.

e) `streamctrl_p->stop()`.This method calls asynchronously the `handle_stop()` method of the `BulkDataCallback`, setting its internal state. Inside this last handle_stop call there is an automatic sleep that waits until all the sent data are received (see section 2.3.4 Timeout).

The `paceData()` method call triggers the same mechanism for passing the data. The only differences occur at point b), where the index 2 is passed in the `send_frame` method, indicating that the data that are going to be received are actual data and not parameters., and at point d), where receive_frame calls the `cbReceive(data)` method of the user callback, passing the data to it.

Finally, inside `stopSend()` the `stopSend(flwn)` method of `BulkdataSender` is called which, in turn, calls the `stop()` method of the `TAO_StreamCtrl` object. This method calls asynchronously `handle_stop()` on the `BulkDataCallback`, which calls the `cbStop()` of the user callback.

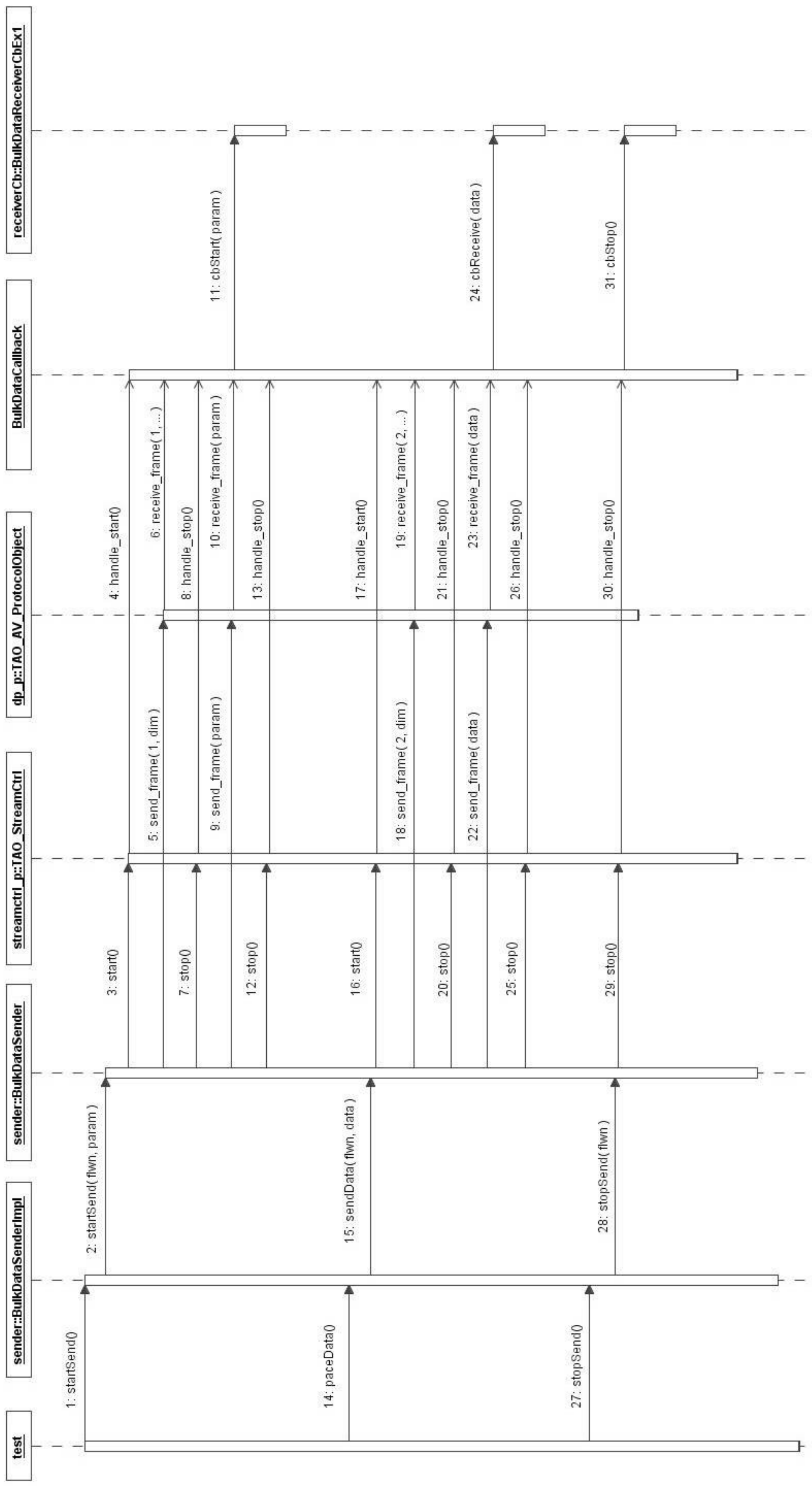

test
sender::BulkDataSenderImpl
sender::BulkDataSender
streamctrl_p::TAO_StreamCtrl
dp_p::TAO_AV_ProtocolObject
BulkDataCallback
receiverCb::BulkDataReceiverCbEx1
1: startSend()
2: startSend(flwn, param)
3: start()
4: handle_start()
5: send_frame(1, dim)
6: receive_frame(1, ...)
7: stop()
8: handle_stop()
9: send_frame(param)
10: receive_frame(param)
11: cbStart(param)
12: stop()
13: handle_stop()
14: paceData()
15: sendData(flwn, data)
16: start()
17: handle_start()
18: send_frame(2, dim)
19: receive_frame(2, ...)
20: stop()
21: handle_stop()
22: send_frame(data)
23: receive_frame(data)
24: cbReceive(data)
25: stop()
26: handle_stop()
27: stopSend()
28: stopSend(flwn)
29: stop()
30: handle_stop()
31: cbStop()

Fig. 13: Sequence diagram for the hand-shake mechanism

7.1 Timeout problem

A timeout problem can occur inside the hand-shake mechanism if the execution time of the receiver callback method `cbReceive` takes too much time.

As described above, when there are data available in the receiver buffer, the ACE Reactor calls the hook method of the handler registered for receiving the data (not visible to the user). This hook method calls the `receive_frame` method of the receiver callback (see point 23 in fig. 10) which, in turn, calls `cbReceive` (point 24 in fig. 10). The return value of the hook method is checked by the ACE Reactor. If it is greater or equal to 0, the Reactor continues dispatching data, calling the hook method again. If the return value is -1 (for example when the connection is closed), the receiver handler is automatically removed from the Reactor, freeing the resources.

If `cbReceive` blocks for a long time before returning (point 24 in fig. 10), the Sender calls `stop()` (point 25 in fig. 10) and then continues, closing the connection. In the meantime, the timeout inside the `handle_stop` expires (point 25 in fig. 10 and point e) in the description of the hand-shake mechanism above), and the Receiver continues closing the connection and deleting the resources. After that `cbReceive` returns, `receive_frame` returns 0 to the hook method which, in turn, returns 0 to the ACE Reactor which calls again the hook method. The hook method calls `receive_frame` but the receiver callback does not exist any more because in the meantime all the resources have been deallocated and the system crashes.

At present this problem is under investigation.

8 References

[1] P. Di Marcantonio, R. Cirami et al., "Transmitting huge amounts of data: design, implementation and performance of the Bulk Data Transfer mechanism in ALMA ACS", ICALEPCS 2005, Geneva, Switzerland, October 2005.